

Rapport de stage

Développement d'un outil de tests pour un logiciel de
contrôle moteur:
GreenT

Antoine de ROQUEMAUREL

M2 Informatique – Développement Logiciel
2015 – 2016

Maître de stage:
Alain FERNANDEZ

Tuteur universitaire:
Jean-Baptiste RACLET

Du 14 Septembre 2015 au 30 Août 2016
Version du June 23, 2016

Je tiens à remercier toutes les personnes m'ayant permis de réaliser ce stage.

En premier lieu, un grand merci à Corinne TARIN pour m'avoir accepté au sein de son équipe.

Je remercie particulièrement Alain FERNANDEZ pour m'avoir suivi et conseillé tout au long de cette alternance tout en partageant son expérience.

Une pensée pour Thierry BOUCHON et Claudine ANDRIEUX, pour la bonne ambiance dans notre bureau, ainsi qu'à toute l'équipe du troisième étage, grâce à qui j'ai passé d'excellents moments au sein de l'entreprise.

Merci à mon tuteur universitaire, Jean-Baptiste RACLET, pour son suivi régulier et ses visites en entreprise.

Enfin, je remercie toutes les personnes m'ayant entouré durant ce stage et aidé à la rédaction ce rapport, à savoir Diane, Ophélie, Clément et Mathieu.

Résumé

Mots-clés: Automobile, Outil, ECU, Contrôle moteur, Tests, Banc de tests, HiL, Debugger, Java, Python

Dans le cadre de ma formation en seconde année de Master Développement Logiciel à l'université Toulouse III, j'ai eu la chance de pouvoir effectuer un an d'alternance. J'ai eu l'opportunité de continuer un projet commencé précédemment : le développement d'un outil de tests de logiciels embarqués.

L'entreprise Continental est une Société Allemande leader de l'automobile possédant plus de 170 000 employés dans le monde. L'entreprise s'occupe aussi bien des calculateurs que de la sécurité automobile, du système d'injection, ...

Pour ma part j'ai travaillé au sein de l'équipe en charge de la mise en place de services de tests. Notamment des tests logiciels, ceci en développant des scripts de tests automatique de non-régression ou d'intégration avant la livraison des projets.

Trois ans avant ce stage, un besoin a été exprimé : pouvoir tester de façon rapide et efficace l'intégration d'un « plugin », un bout de code sous forme binaire, au sein des applicatifs d'un calculateur de contrôle moteur. La mission de l'équipe *Tests & Automation Service* est de permettre de tester la bonne intégration de ce plugin avec les logiciels Continental. Pour cela le développement d'un outil de tests est nécessaire.

Au début de mon alternance, cet outil appelée GreenT, avait déjà était bien avancée, en partie lors de mes stage précédent. Cependant, à mon arrivée un module devait être refondu afin de pouvoir livrer une première version de l'outil. Une fois celle-ci livrée, de nouvelles fonctionnalités ainsi que du support de la maintenance étaient nécessaires. C'est donc dans ce contexte que j'ai travaillé durant cette année, ceci afin d'améliorer au maximum l'outil, tout en accompagnant les utilisateurs.

Cet outil a pour but d'avoir des tests les plus exhaustifs et efficaces possibles. Afin de pouvoir tester la bonne intégration du plugin, le client fourni un fichier Excel appelé Walkthrough contenant la liste des variables du plugin avec toutes leur spécifications. Le testeur va ajouter des colonnes à ce fichier afin de spécifier le fonctionnement du test, notre plateforme sera ensuite capable d'analyser le fichier, et de générer les cas de tests qui s'exécuteront à distance sur un ou plusieurs bancs de tests : ils simulent un environnement véhicule autour du contrôleur afin de vérifier ses réactions en fonction des différentes conditions qui peuvent arriver.

Mon travail durant ce stage aura été bénéfique, pour l'entreprise grâce à mes connaissances de l'outil, au développement de nouvelles fonctionnalités, à mes corrections de bugs et au support que j'ai effectué. Mais aussi personnellement, d'un point de vue technique, en trouvant des solutions à des problèmes. Et d'un point de vue humain grâce au travail en équipe, aux comptes rendus réguliers qui m'ont permis d'apprendre à synthétiser mon travail.

Le projet n'est pas terminé, celui-ci nécessitant encore de la maintenance et du développement de nouvelles fonctionnalités. Cependant, avec ces deux ans de travail, j'aurai eu l'opportunité de voir un cycle logiciel complet, allant de l'analyse des besoins jusqu'à l'exploitation de l'outil, ce qui sera un atout pour ma future carrière professionnelle.

Abstract

Keywords: Automotive, Tool, ECU, Control engine, Tests, Workbench, HiL, Debugger, Java, Python

During my second year of Master in computer science at Toulouse III University – Paul Sabatier, I had the opportunity to do an internship. I chose to work in continuity of my 1st year, the development of a tool for embedded tests, in Continental, an automotive company.

Continental is a leading German automotive manufacturing company who has 174 000 employees all around the world. The company is specializing in tires, brake systems, automotive safety and powertrain. I worked in the team that was in charge of software tests, with the development and configuration of scripts. Those scripts are for regressions tests, and in my case, integration tests.

Three years before this internship, a need was been expressed: being able to quickly and efficiently test the integration of a plugin, meaning a piece of code in binary. This plugin is for an engine control software. The mission of the team Tests & Automation Service is helping the proper integration of this plugin with Continental software. For this purpose, the development of a test tool is required.

The development of this tool, called GreenT, began three years ago. GreenT needs bugs corrections and features development for production deployment. So, I worked in this context during one year, and in january, the release 1.0 was deployed.

The goal of this tool was to provide the most exhaustive and effective tests possible. For this purpose, the customer provides a specification file, called Walkthrough, wich contains the specification of all plugin's variables. Testers will add columns in this file for test specification. After this, GreenT will be able to parse and generate tests case that will be executed remotely on one or several workbenches. A workbench can simulate a vehicle environment around an ECU to check reactions.

I developed some features in this project: the improvement of reports generation, which is required for delivery, and systems for opening GreenT to most of projects. In addition, since january, I support users which writing tests.

So my work during this internship was beneficial for the company, my knowledge about GreenT allowing me to help fixing bugs, and I was able to develop some new features, and help to delivery the first operational version of the tool. It was also beneficial for me, from a technical point of view by designing solutions for particular problems, and from a human point of view by improving my teamwork skills thanks to the meetings we did to talk about the development.

This project is not completed yet, it required maintenance and development of new features. However, this two years of development have shown me a complete software lifecycle, from needs analyze to exploitation of tool. It will be an asset for my future professional career.

Introduction

Dans le cadre de ma formation en seconde année de Master Développement Logiciel à l'université Toulouse III – Paul Sabatier, j'ai eu la possibilité d'effectuer un contrat de professionnalisation en alternance d'une durée d'un an.

Attiré par le monde de l'entreprise et désireux de gagner en expérience, j'ai pu continuer un projet commencé précédemment lors de mes stages de licence et de M1, dans l'entreprise Continental Automotive : le développement d'un outil de tests de logiciels embarqués.

Ce projet a pour but d'aider une équipe de Continental. Celle-ci a des difficultés au niveau de l'intégration d'un plugin dans un logiciel de contrôle moteur. Celui-ci possède des milliers de variables interfaces et est donc compliqué à tester. Afin d'aider cette équipe, un outil permettant d'effectuer des tests automatiques est en développement depuis Janvier 2014 : *GreenT*.

Dans un premier temps, afin que ce projet puisse disposer d'une première version de production, il a été nécessaire de refondre le système de prononciation des verdicts de tests.

Dans un second temps, j'ai effectué du support, de la formation et de la maintenance sur le projet, tout en continuant le développement afin d'ajouter de nouvelles fonctionnalités.

Ayant connu les prémices de ce projet, et afin d'avoir un aperçu de celui-ci sur la durée, allant de sa conception jusqu'à son exploitation, le sujet du stage était particulièrement intéressant. En outre, celui-ci est en parfaite adéquation avec mon projet professionnel, le développement et la conception de logiciels destinés à l'industrie. En effet, ce projet est au cœur du problème d'ingénieur logiciel, par la problématique que celui-ci essaye de régler : la manière dont nous pouvons tester un logiciel.

J'ai travaillé au sein de l'équipe *Test & Automation Service*, je vais ainsi vous présenter en quoi le développement de cet outil est nécessaire à l'équipe en charge des tests de ce plugin.

Dans une première partie nous présenterons le contexte dans lequel j'ai travaillé, avec l'entreprise Continental et plus particulièrement l'équipe *Tests & Automation Service* (chapitre 1), puis nous verrons le problème que posent actuellement les tests de ce plugin (chapitre 2), nous aborderons ensuite la manière dont nous nous sommes organisés pour le développement (chapitre 3) avant de présenter la solution qui a été développée (chapitre 4) et comment j'ai contribué à ce projet (chapitre 5).

Contents

Remerciements	3
Résumé	5
Abstract	6
Introduction	7
1 Continental	11
1.1 Organisation de l'entreprise	11
1.2 Le contexte de l'équipe TAS	14
2 Le problème	17
2.1 Composant logiciel tiers	17
2.2 Les tests du « plugin » Ford	17
2.3 La solution : <i>GreenT</i>	19
3 Organisation du développement	21
3.1 L'équipe de développement	21
3.2 Les méthodes de travail	21
3.3 Les outils de développement	22
4 <i>GreenT</i> : fonctionnement général	25
4.1 Le fichier TestPlan	25
4.2 Fonctionnement Global	26
4.3 Les fonctionnalités du client	27

4.4	Les fonctionnalités des serveurs	31
5	Ma collaboration au projet	33
5.1	La difficulté de productions de rapports	33
5.2	L'arrivée des projets multi-cœurs	35
5.3	Ouverture de l'outil à d'autres projets	37
5.4	La synchronisation des traces	38
6	Bilans	43
6.1	Bilan pour Continental	43
6.2	Bilan personnel	44
A	Acronymes et Glossaire	47
B	Index	49
C	Références	51
C.1	Documentations	51
C.2	Livres	51
C.3	Cours magistraux	52
C.4	Sites Web et forums	52
D	Exemple de rapport généré par GreenT	53
E	Release note	55
F	Exemples de fichiers générés	57
F.1	Exemple de StimScenario	57
F.2	Exemple de GreenTTest	58
G	List of Figures	61
H	Liste des codes sources	63

Mon contrat de professionnalisation s'est déroulé au sein de l'entreprise Continental. Cette entreprise est, sur les dernières années, entre première et deuxième équipementier automobile mondial par le volume de vente.

Sommaire

1.1	Organisation de l'entreprise	11
1.2	Le contexte de l'équipe TAS	14

Ce chapitre présente rapidement le groupe Continental et plus particulièrement l'équipe *Tests & Automation Service* qui m'a accueilli pour ce stage.

1.1 Organisation de l'entreprise

1.1.1 Le groupe Continental

Continental est une entreprise allemande fondée en 1871 dont le siège se situe à Hanovre. Il s'agit d'une Société Anonyme (SA) dont le président du comité de direction est le Dr. Elmar DEGENHART depuis le 12 août 2009. Le groupe Continental est constitué de cinq divisions intervenant sur le marché des pneus (*Rubber*) et de l'électronique automobile (*Automotive*), ces divisions vous sont détaillées figure 1.4.

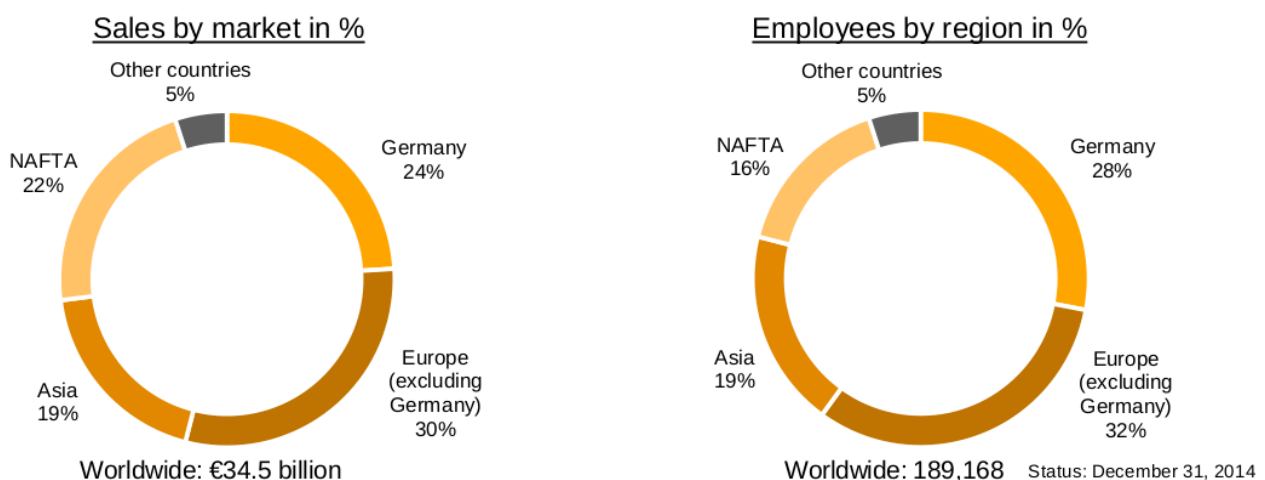


Figure 1.1: Chiffre d'affaire et nombre d'employés (Année 2014)¹

¹NAFTA : North American Free Trade Agreement

En 2014, l'entreprise comptait plus de 189 000 employés dans le monde, comme le montre la figure 1.1, répartis dans 317 sites et 50 pays différents, dont la répartition est détaillée figure 1.2. Avec un chiffre d'affaire de 34.5 milliards d'euros au total, Continental est le numéro un du marché de production de pneus en Allemagne et est également un important équipementier automobile.

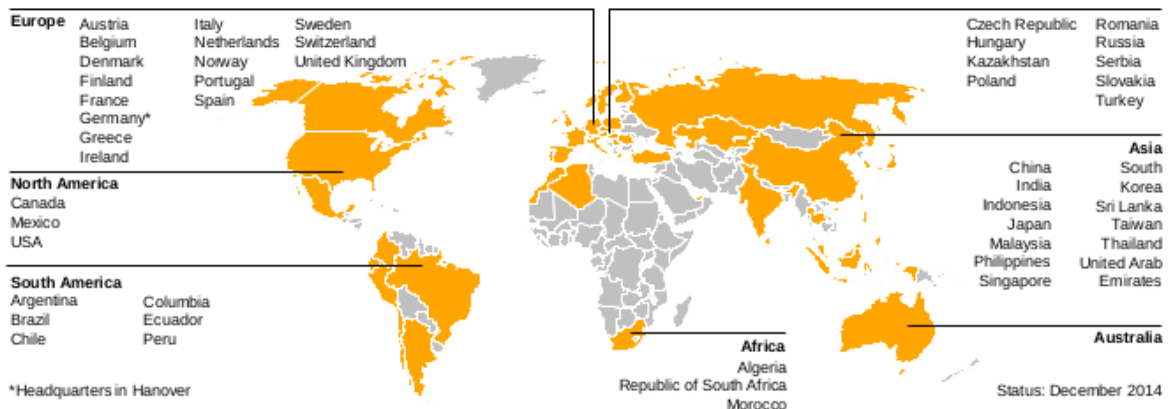


Figure 1.2: Répartition du groupe Continental dans le monde

1.1.2 Histoire de l'entreprise

Continental est fondée en 1871 comme société anonyme sous le nom de « *Continental-Caoutchouc-und Gutta-Percha Compagnie* » par neuf banquiers et industriels de Hanovre (Allemagne).

Continental dépose l'emblème du cheval représenté sur la figure 1.3, comme marque de fabrique à l'Office impérial des brevets de Hanovre en octobre 1882. Ce logo est aujourd'hui encore protégé en tant que marque distinctive.



Figure 1.3: Logo de Continental

Le fabricant de pneus allemand débute son expansion à l'international en tant que sous-traitant automobile international en 1979, expansion qu'il n'a cessé de poursuivre depuis.

Entre 1979 et 1985, Continental procède à plusieurs rachats qui permettent son essor en Europe, celui des activités pneumatiques européennes de l'américain *Uniroyal Inc.* et celui de l'autrichien *Semperit*.

En 1995 est créée la division « *Automotive Systems* » pour intensifier les activités « systèmes » de son industrie automobile.

La fin des années 1990 marque l'implantation de Continental en Amérique latine et en Europe de l'Est.

En 2001, pour renforcer sa position sur les marchés américain et asiatique, l'entreprise fait l'acquisition du spécialiste international de l'électronique *Temic*, qui dispose de sites de production en Amérique

et en Asie. La même année, la compagnie reprend la majorité des parts de deux entreprises japonaises productrices de composants d'actionnement des freins et de freins à disques.

En 2004, le plus grand spécialiste mondial de la technologie du caoutchouc et des plastiques naît de la fusion entre *Phoenix AG* et *Conti'Tech*.

En juillet 2007, Continental réalise sa plus grosse opération en rachetant le fournisseur automobile *Siemens VDO Automotive*. Ce rachat a permis à l'entreprise de multiplier son chiffre d'affaire par 2.5, passant ainsi de 13 milliards d'euros à plus de 34.5 milliards d'euros (chiffre de 2014).

En mai 2015, Continental annonce le rachat pour 600 millions d'euro de la branche automobile du groupe finlandais Elektrobit, afin de diversifier sa gamme.

Enfin, en Septembre 2015, Continental rachète l'activité contrôle moteur de Valeo, correspondant à une centaines de personnes.

1.1.3 Activités des différentes branches

Chassis & Safety	Powertrain	Interior	Tires	ContiTech
Vehicle Dynamics	Engine Systems	Instrumentation & Driver HMI	PLT, Original Equipment	Air Spring Systems
Hydraulic Brake Systems	Transmission	Infotainment & Connectivity	PLT, Repl. Business, EMEA	Benecke-Kaliko Group
Passive Safety & Sensorics	Hybrid Electric Vehicle	Intelligent Transportation Systems	PLT, Repl. Business, The Americas	Compounding Technology
Advanced Driver Assistance Systems (ADAS)	Sensors & Actuators	Body & Security	PLT, Repl. Business, APAC	Conveyor Belt Group
	Fuel & Exhaust Management	Commercial Vehicles & Aftermarket	Commercial Vehicle Tires	Elastomer Coatings
			Two Wheel Tires	Fluid Technology
				Power Transmission Group
				Vibration Control

PLT – Passenger and Light Truck Tires

Figure 1.4: Structure de Continental

Comme on peut le voir sur la figure 1.4, Continental est composée de cinq divisions. Ces dernières se chargent de développer et produire des équipements répondant aux besoins des clients. Pour cela elles sont composées de *Business Units* qui ont chacune une activité bien particulière dans leur domaine de compétence.

Durant mon stage, je travaillais au sein de *P-ES* :

Division *Powertrain* S'occupe essentiellement du contrôle moteur, au niveau logiciel et matériel avec l'ECU²

Business Unit *Engine Systems* Chargée de produire les équipements nécessaires au contrôle moteur tels que des calculateurs ou des injecteurs.

²Engine Control Unit, Unité de calcul du contrôle moteur

1.2 Le contexte de l'équipe TAS

J'ai travaillé dans l'équipe en charge des tests au niveau système ou logiciel dirigée par Corinne TARIN. Cette équipe doit aider à la vérification et la validation des programmes de contrôle moteur en fournissant des services de tests.

1.2.1 Le besoin

Le calculateur du contrôle moteur d'une voiture est un dispositif très important et à haut risque, en effet, une défaillance peut provoquer la mort de plusieurs personnes. Le programme d'une voiture comporte ainsi des fonctions dites « *safety* » tel que le l'accélération, le freinage, le régulateur... Ainsi, le test est indispensable dans ce domaine, et doit être robuste.

Le test des logiciels de contrôle moteur se fait aujourd'hui :

- Soit « à la main » pour certains cas de tests.
- Soit à l'aide de scripts de test Python, écrit manuellement comme présenté section 1.2.3.

Cependant, la taille des logiciels à tester est devenue particulièrement importante (Plusieurs milliers de variables, dans plus de 10 000 pages de spécification...). Cela appelle à une automatisation plus forte des tests afin d'augmenter fortement la vitesse et la quantité de tests pour éliminer le maximum de bogues.

C'est dans ce contexte que l'équipe TAS intervient, c'est ainsi que je participe au développement d'un outil permettant d'automatiser des tests d'intégration pour les équipes projet travaillant pour Ford.

1.2.2 Les tests automatisés

Pour ma part, j'opérais dans la partie tests automatiques. Cette « sous-équipe » possède deux missions :

- Le développement, l'exécution et la maintenance de scripts de tests de non-régression³. Ces scripts de tests s'exécutent sur bancs HiL⁴, avant la livraison des projets. Vous trouverez plus d'explications sur ce dispositif section 1.2.3.
- Le développement et la maintenance d'outils logiciels, dans mon cas, celui-ci à pour but d'améliorer la couverture et la qualité des tests. Ils permettra aux développeurs de vérifier facilement et correctement leur travail, particulièrement dans le cadre de tests d'intégration.

³Aussi appelés FaST : **F**unctions and **S**oftware **T**esting

⁴**H**ardware **I**n the **L**oop

Les verdicts

Une fois un test automatisé exécuté, l'outil de tests va prononcer un verdict. Ces verdicts peuvent être :

- vert, si le code testé est estimé correct,
- rouge, si le code testé ne correspond pas aux spécifications.

Lorsque l'on effectue des tests, le but du testeur est de trouver des bugs. Ainsi, un positif est l'apparition d'un bug, et donc un faux positif signifie que la plateforme soulève des bugs, qui sont inexistantes

1.2.3 Les outils de tests

Afin d'effectuer son travail, l'équipe TAS possède différents outils de tests. D'une part au niveau matériel avec des bancs de tests, mais aussi logiciel avec un outil écrit en Python.

Les bancs de tests

Afin de tester au mieux les programmes du contrôle moteur développés, ceux-ci sont d'abord testés via des simulateurs d'environnement véhicule. Ce simulateur permet de vérifier le programme avant d'effectuer des tests sur véhicule. Ces tests se font sur table dans un premier temps, pour deux raisons principales :

- D'une part, les tables sont plus facilement accessibles qu'un véhicule d'essai pour les équipes logiciel
- D'autre part, les tables possèdent plus de moyens afin d'observer finement l'ECU

Comme vous pouvez le voir figure 1.5, un banc de tests est composé d'un ordinateur, du calculateur appelé ECU pour *Engine Control Unit* et d'au moins deux équipements aidant aux tests.

Les deux équipements étant les suivants :

Le HiL Le *Hardware in the Loop* est un simulateur d'environnement véhicule. Ainsi l'ECU est branché sur le HiL et se comporte de la même manière que s'il était embarqué dans une voiture. Le HiL quant à lui est chargé d'envoyer les bons stimuli sur les pins de l'ECU, tel que l'injection, la vitesse de rotation du moteur, le starter ...

Le Debugger Cet appareil est connecté au microcontrôleur de l'ECU via un port JTag. Il peut communiquer avec celui-ci afin d'effectuer différentes opérations. Tel que flasher le logiciel à tester, mettre des points d'arrêts sur le code, lire des variables, les modifier, changer des calibrations, ...

Les différents équipements, ou *devices*, que nous pouvons voir sur la figure 1.5 sont ceux avec lesquelles notre nouvelle outil va communiquer dans le but d'effectuer des tests automatiques.

Remarque Un certain nombre d'équipes projet chez Continental utilise un troisième équipement qui n'est pas représenté ici, parce que notre outil ne s'en sert pas. Cet équipement, nommé INCA, va interagir sur l'ECU via un bus CAN^a.

Lors de mon dernier développement, j'ai ajouté un nouvel équipement, un analyseur logique. Cet équipement se branche sur le debugger et permet d'observer des signaux numériques. Son utilisation et son besoin sont détaillés section 5.4.

^aController Area Network

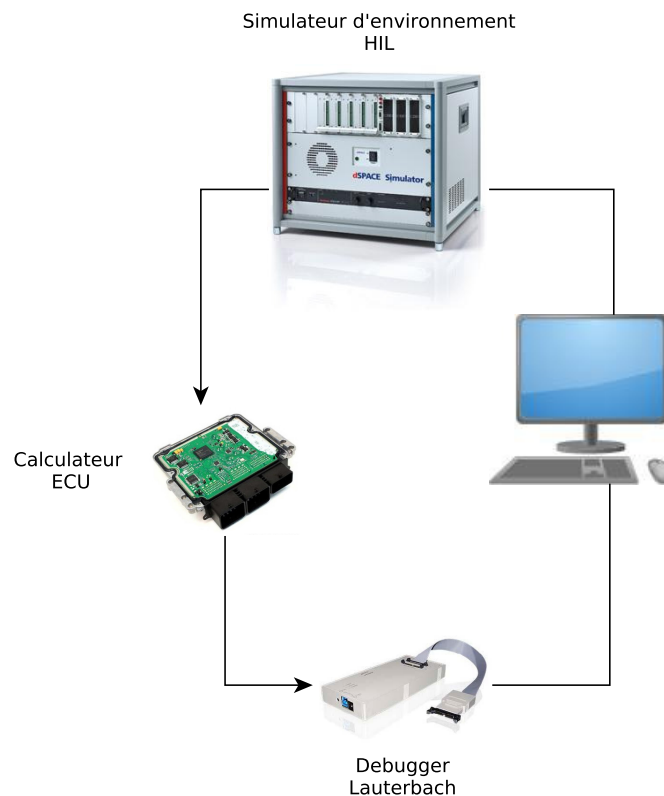


Figure 1.5: Fonctionnement d'une table de tests : HiL DSpace, Debugger et ECU

La plateforme TA3



Actuellement, les équipes de tests disposent d'une plateforme appelée TA3. Celle-ci est une bibliothèque de classes écrite en Python. Jusqu'à présent, pour chaque objectif de test, il fallait écrire un script Python utilisant la TA3. Ces scripts pilotent le banc HiL et le debugger afin d'envoyer des stimuli à l'unité de contrôle moteur et de vérifier que les réactions de celui-ci sont conformes aux spécifications de test.

Cependant, cette plateforme pose un certain nombre de problèmes qui rend son utilisation difficile. D'une part, elle renvoie un trop grand pourcentage de faux-positifs faisant perdre du temps aux testeurs. D'autre part, elle ne prend pas en compte certains besoins apparus récemment comme un système permettant de flasher automatiquement les ECU, ce qui permettrait de scripter un test qu'on lancera plus tard et de gagner du temps, ou la possibilité de vérifier la fréquence de mise-à-jour de la production de variables, afin de contrôler le temps réel du calculateur.

Afin d'améliorer cette situation, l'équipe *Tests & Automation Service* développe un nouvel outil de tests qui a fait l'objet de mon alternance : GreenT.

Depuis longtemps, l'entreprise avait un problème afin d'effectuer des tests d'intégrations, notamment pour les projets à destination de Ford. Les tests demandaient du temps et de l'argent à l'équipe en charge de ces tests. Ainsi, deux ans avant mon stage de M2, un projet visant au développement d'un nouvel outil de test, GreenT, a été initié.

Sommaire

2.1	Composant logiciel tiers	17
2.2	Les tests du « plugin » Ford	17
2.3	La solution : <i>GreenT</i>	19

2.1 Composant logiciel tiers

Étant donné la taille grandissante des programmes informatiques, il est de plus en plus rare qu'une seule et unique entité effectue le développement d'un logiciel.

C'est ainsi que chez Continental, un certain nombre de composants des calculateurs ne sont pas développés en interne. Ce concept est appelé composant logiciel tiers, ou *Third-Party Software*.

La principale difficulté de ce mode de fonctionnement est l'intégration, une spécification exhaustive est indispensable afin de pouvoir connecter les différentes interfaces du composant avec le reste du projet.

C'est avec ces contraintes que travaillent plusieurs équipes chez Continental et notamment l'équipe en charge du développement des différents logiciels de contrôle moteur à destination de Ford. C'est ainsi que cette équipe doit intégrer un plugin en tant que logiciel tiers au sein des logiciels Continental.

2.2 Les tests du « plugin » Ford

Comme dit précédemment, Continental ne développe pas l'intégralité du logiciel pour Ford, une partie étant fournie par le client sous forme de « plugin ». Ce plugin est un fichier binaire qui est chargé directement dans la mémoire flash de l'ECU sans que Continental n'ait accès au code source. Seule une description des interfaces est fournie. Le *plugin* est supposé correct, et le tester n'est pas de notre ressort. Cependant, celui-ci va être interfacé avec les logiciels Continental : il est indispensable de vérifier que les deux parties fonctionnent ensemble lors de l'intégration.

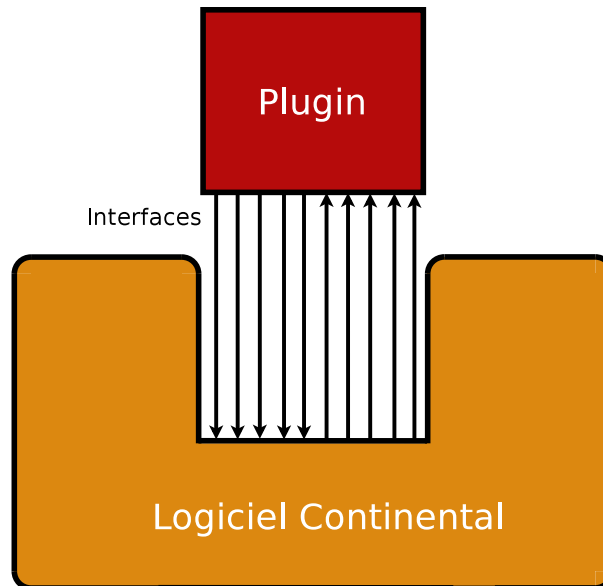


Figure 2.1: Interfaces du plugin avec le logiciel de Continental

Le fichier de spécification est un fichier Excel, fourni par le client. Ce fichier, appelé *Walkthrough*¹, contient la liste de toutes les variables du plugin avec toutes leur spécifications. Il contient environ 1200 variables différentes.

Il est impensable de tester le fonctionnement d'autant de paramètres manuellement. Ainsi l'équipe en charge de tester cette intégration effectue des tests de différence d'une version à l'autre : seules les variables ayant pu être impactées par une *release* seront testées, il est supposé que le fonctionnement des autres variables reste inchangé. Ce type de test est appelé *Delta Test*.

Deux problèmes se posent à cette méthode :

La fiabilité des tests manuels Le test des seules différences ne permet pas nécessairement de détecter tous les problèmes. De plus, une tâche répétitive peut entraîner des erreurs humaines.

Le temps de tests Même en ne testant qu'une partie des variables, cela prend un temps considérable, il faut compter environ une semaine pour l'ensemble des tests.

Or, les tests s'effectuent sur les bancs de tests comme expliqué section 1.2.3, ces équipements permettent de simuler un environnement véhicule du contrôleur moteur comme l'utilisation de la clé de démarrage, la tension de la batterie, la vitesse de rotation du moteur, ... Ceux-ci sont peu nombreux dans l'entreprise en raison de leur coût, leur disponibilité est donc compliquée. Il serait intéressant de pouvoir lancer des tests automatisés durant la nuit par exemple afin d'optimiser au maximum leur utilisation.

2.2.1 Le nombre de versions

Un autre problème des projets Ford est la variance. En effet, dans le cadre du développement du moteur Panther, le programme du calculateur est destiné à un grand nombre de véhicules de

¹Ce fichier est expliqué plus en détail section 4.1

gammes différentes. Ainsi, l'équipe de développement a choisi d'avoir un système de branches de programme, c'est-à-dire une branche principale, FP2 pour Panther Phase 2, dérivée pour l'ensemble des projets concernés, FPC, FPD, FPF, ...

Or, pour chacun des projets, il peut y avoir cinq ou six livraisons différentes, ce qui fait un nombre total de livraisons conséquent. Or tous les projets sont basés sur la même branche principale, des tests rédigés pour la branche principale pourrait être réutilisé et exécutés de nouveau pour les branches dérivées, afin d'effectuer des tests de non régression. Cette action est particulièrement compliquée manuellement, c'est en ça que l'équipe a besoin d'un outil leur simplifiant la tâche, et assurant la fiabilité de leurs programmes.

2.3 La solution : *GreenT*

Pour répondre aux besoins de l'équipe Ford, une solution a été pensée en étudiant leurs besoins : le développement de *GreenT*

Remarque Le nom de *GreenT* provient de la contraction de *Green* et *Test*.

En effet, un test est signalé correct si celui-ci est vert, or le but de notre plateforme est d'automatiser des tests et qu'à la fin de l'exécution, tout ceux-ci soient verts.

2.3.1 Utilisation de l'outil pour les tests d'intégration du plugin Ford

Depuis Janvier dernier, *GreenT* est sorti en version 1.0, ainsi cette solution permet de tester le plugin pour les projets Ford facilement et de façon efficace.

Pour cela, un testeur de l'équipe a ajouté des colonnes dans le document *Walkthrough*, afin de spécifier la manière de tester les variables. L'outil est capable d'analyser le document *Walkthrough*, et de générer les tests automatiques. Il est ensuite possible de planifier son exécution, celle-ci peut prendre deux heures pour 270 tests. Une fois l'exécution terminée, le testeur a tous les résultats de ces tests, il peut ainsi regarder les rapports détaillés afin de corriger les éventuels problèmes.

Ces tests s'effectuent sur des variables enregistrées lors de stimulation de l'ECU, afin de vérifier que celui-ci réagit de façon approprié.

Cet outil permettra ensuite de tester facilement la dizaine de projets Ford, et une fois le test d'une variable spécifié, il n'est plus nécessaire de le réécrire. À chaque nouvelle version du logiciel, ou *release*, il suffit de relancer les tests : l'équipe n'a à faire le travail qu'une fois en début de projet, ensuite la réutilisation sera possible, les projets seront testés plus rapidement, plus efficacement, et plus souvent.

2.3.2 Utilisation de l'outil pour Les autres projets

À court terme, cet outil pourrait être utilisée pour les projets d'autres clients tel que Renault, afin d'effectuer là aussi des tests d'intégration. Il était donc nécessaire de concevoir un outil qui puisse évoluer facilement, et puisse avoir un fichier de spécification en entrée qui soit légèrement différent d'un client à l'autre.

Afin que notre outil puisse fonctionner sur le maximum de projets différents, et qu'il puisse également servir à effectuer des tests d'intégration indépendamment du cadre d'un *third party software*, j'ai mis en place un fichier de spécification en entrée totalement générique. Plus d'information sur ce fichier section 5.3.

Renault a cependant des besoins légèrement différents de ceux de Ford, pouvoir exprimer un test en fonction de variables ECU, tel que Ford, mais également de symboles HiL. Une nouvelle fonctionnalité est donc en développement comme présenté section 5.4.

3

Organisation du développement

Étant donné la complexité du projet et son importance, une organisation réfléchie est indispensable. Autant d'un point de vue humain, avec une gestion de projet et une gestion de l'équipe, que technique en utilisant certaines technologies nous aidant dans la tâche. Nous allons voir l'organisation qui a été mise en place afin d'être le plus efficace possible.

Sommaire

3.1	L'équipe de développement	21
3.2	Les méthodes de travail	21
3.3	Les outils de développement	22

3.1 L'équipe de développement

Au cours de mon stage, deux développeurs ont travaillé sur le projet *GreenT* : Alain FERNANDEZ, chef d'équipe et membre de l'équipe *Tests & Automation Service*, et moi-même.

Jusqu'à la livraison de la première version de l'outil en Janvier, nous avons travaillé en coordination afin de finir le développement de l'analyse et de la production des rapports détaillés, celui-ci étant bloquant pour pouvoir livrer la première version de l'outil.

Une fois la livraison de l'outil terminée, nos méthodes de travail ont été légèrement modifiées. Alain a suivi l'avancée du projet, et notamment mon travail, et était présent aux différentes réunions avec nos clients internes. Cependant, celui-ci n'a effectué que peu de développement, bien que son expertise restait indispensable pour un certain nombre de questionnements sur la continuation du projet. Quant à moi, j'ai effectué du support utilisateur ainsi que de la maintenance, notamment en vue de l'utilisation de l'outil sur les différents projets Ford. En parallèle de ce développement, j'ai développé de nouvelles fonctionnalités en vue de l'utilisation de l'outil à d'autres utilisateurs que Ford.

3.2 Les méthodes de travail

Afin de pouvoir continuer le développement de nouvelles fonctionnalités tout en conservant la base de code en production, nous avons utilisé un workflow Git particulier, permettant d'avoir une branche master correspondant à la branche actuellement en production, et une branche par *release*. Chaque branche de release pouvant éventuellement avoir des branches distinctes par fonctionnalités.

3.2.1 La documentation

Deux types de documents sont importants dans le cadre d'un développement en production, les documents à destination de l'utilisateur, et les documents destinés à la documentation interne.

Les documents utilisateurs, rédigés en Anglais, sont mis-à-jour pour chaque version de l'application et sont livrés directement avec l'exécutable :

- Le manuel utilisateur, contenant le fonctionnement global de l'outil, la syntaxe du langage, la configuration de celui-ci ainsi que son utilisation.
- Une *release note*, spécifique à une version, qui permet de connaître les changements de la version, et les éventuels problèmes de compatibilité.

La documentation interne contient plusieurs documents, rédigés en français au vu de leurs destination, sont mis-à-jour régulièrement en cas de changement de l'outil.

- Les documents d'exigences, utilisateur ou système, ces documents nous servent de référence quant aux besoins des utilisateurs, et au fonctionnement du système. En cas de nouvelle fonctionnalité, ceux-ci doivent être mis-à-jour
- Les différents modules de l'outil possèdent chacun leurs documents de conception et de fonctionnement, ceux-ci sont mis-à-jour dans la mesure du possible, et certains modules peuvent posséder de la dette technique, dette que j'essaye de régler au fur et à mesure du développement, lorsque j'ai un peu de temps.
- La documentation sur des outils interne, certains outils que nous utilisons, nous avons rédigés des documents permettant de comprendre rapidement le fonctionnement celui-ci et son utilisation dans GreenT. C'est le cas par exemple pour Git, Apache Thrift, Antlr ou Trace32. Ces documentations étaient rédigées par la personne ayant le plus de connaissance ou d'expérience de l'outil, j'ai ainsi rédigé une documentation sur Git, Antlr, Ant ainsi que le fonctionnement du Powerprobe Lauterbach.

3.2.2 Les livraisons de l'outil

Version 1.0.0 Première version fonctionnelle de l'outil, cette version à été livrée en Janvier 2016

Version 1.1.0 Version contenant différentes corrections de bogues, ainsi que l'utilisation sur ECU multi-core, sa release note est disponible en annexe E.

Version 1.2.0 Cette version n'a pas encore été livrée, mais le sera avant la fin de mon contrat. Celle-ci contiendra la synchronisation des traces qui est nécessaire aux tests Renault.

3.3 Les outils de développement

Afin de travailler de façon efficace, nous avons utilisé des outils aidant au développement. Ces outils ont été définis au début du projet, et n'ont pas évolués depuis.

3.3.1 Java

À mon arrivée, la partie client de notre plateforme était développée en Java dans sa version 6.0, Java nous permettant d’avoir un langage fortement typé, très puissant au niveau du paradigme Objet, connu de l’équipe, assez simple de déploiement et multiplateforme.



Une de mes collaboration a été le passage à Java 8 nous permettant d’utiliser toute la puissance de Java, et d’avoir une plateforme qui soit à jour au niveau technologique.

3.3.2 Git



Nous avons utilisé *Git* afin de faciliter le travail collaboratif d’une part, et de versionner le code du logiciel d’autre part. Git permet de fusionner les modifications de plusieurs développeurs, tant que nous ne modifions pas le même fichier en même temps. Ainsi, la fusion de nos modifications était faite automatiquement.

De plus, à chaque nouvelle modification, un « commit », permet de créer un point de restauration : il est alors possible de récupérer n’importe quelle version du logiciel depuis son commencement. Nous y insérons un message clair expliquant ce qui a été fait, cela permet aux autres développeurs de l’équipe de se tenir au courant de l’avancement.

3.3.3 Thrift et client-serveur

Notre plateforme fonctionne avec une architecture client-serveur, un client et deux serveurs. Le client écrit en Java, un serveur utilise Python et le second est lui aussi en Java. Afin de faire communiquer les deux parties de notre application, nous avons utilisé *Apache Thrift*. Il s’agit d’une bibliothèque ayant pour but les communications réseaux inter-langage, dans le même principe que le protocole RMI¹.

Ainsi, nous avons rédigé un fichier spécifiant les interfaces de notre serveur, c’est-à-dire les méthodes que nous souhaitons appeler en réseau. Une fois ce « contrat » rédigé, il faut demander à Thrift de générer le code du serveur – génération qu’il est possible de demander en C, C++, Python, Java, C#, PHP, Ruby, ... – et du client. Côté client, le service s’utilise directement, côté serveur, il faut implémenter une interface afin que notre service effectue les bonnes instructions. C’est donc le code généré qui va se charger de l’abstraction réseau.

3.3.4 Eclipse

Nous développons tous sous le même environnement de développement Eclipse, avec le plugin *Git* et le plugin *PyDev*. Le plugin Git permet d’avoir des outils



¹Remote Method Invocation

aidant à la résolution d'éventuels conflits et le plugin PyDev permet de développer avec l'interpréteur et la coloration syntaxique Python.

3.3.5 Antlr



Pour les besoins de notre plateforme, nous avons créé notre propre langage de test. Ce langage est assez riche, et la création d'un parser adéquate aurait pu être particulièrement longue. Afin de nous faire gagner le maximum de temps, nous avons utilisé Antlr4. *Another Tool For Language Recognition* est un programme permettant de générer automatiquement un parser pour un langage donné. Ainsi, nous avons rédigés notre grammaire, Antlr quant à lui s'est chargé de nous générer un arbre de parcours syntaxique. À notre charge d'effectuer les bonnes actions durant le parcours de notre langage en spécialisant les classes générées par Antlr.

3.3.6 UML et Enterprise Architect

Nous avons travaillé avec la norme UML² 2 afin de concevoir la plateforme, en utilisant particulièrement des diagrammes de classes, mais aussi des diagrammes de cas d'utilisation ou d'activité.

Pour dessiner ces diagrammes, et les noter dans la documentation, nous les pensions d'abord sur tableau blanc, mais ensuite nous avons besoin d'un outil puissant afin de les dessiner sur informatique. Pour cela nous avons utilisé *Enterprise Architect*, un logiciel propriétaire permettant de créer tous les diagrammes de la norme UML 2.



3.3.7 SQLite



SQLite est un moteur de base de données relationnelle. Sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégré aux programmes, la base de données étant stockée dans un simple fichier.

Nous nous en sommes servis afin de pouvoir stocker les différentes informations d'un test, ceci afin de pouvoir redémarrer une exécution grâce à cet état intermédiaire conservé en base de données.

3.3.8 L^AT_EX

Afin de rédiger ce rapport, et le diaporama de soutenance, j'ai utilisé L^AT_EX, un langage et un système de composition de documents fonctionnant à l'aide de macro-commandes. Son principal avantage est de privilégier le contenu à la mise en forme, celle-ci étant réalisée automatiquement par le système une fois un style défini.



²Unified Modelling Language

4

GreenT : fonctionnement général

Comme nous l'avons montré dans le chapitre 2, l'entreprise a besoin d'un nouvel outil aidant aux tests d'intégration : *GreenT*.

Nous allons donc voir le développement et la conception de cet outil de tests.

Au début de mon stage, le projet ayant deux ans, les fonctionnalités développées ci-dessous étaient déjà faites. Je suis cependant intervenu sur la plupart d'entre elles soit pour des corrections de bogues d'une part, soit à des fins d'améliorations d'autre parts.

Avant de présenter mon travail, que vous trouverez chapitre 5, il est nécessaire de présenter le fonctionnement général de cette plateforme afin d'en avoir une vue d'ensemble.

Sommaire

4.1	Le fichier TestPlan	25
4.2	Fonctionnement Global	26
4.3	Les fonctionnalités du client	27
4.4	Les fonctionnalités des serveurs	31

4.1 Le fichier TestPlan

Le fichier TestPlan est le fichier d'entrée du test, permettant de connaître la manière dont nous devons tester un logiciel. Initialement, GreenT n'était prévu que pour utiliser le fichier Walkthrough, un fichier spécifique à Ford.

Celui-ci sera fourni par la personne en charge des tests, c'est un fichier au format Excel qui contient les informations sur chacune des variables à tester. Il contient ainsi un très grand nombre de colonnes, bien que seule une partie de celles-ci nous intéressent. Certaines colonnes ont été remplies par le fournisseur du plugin, d'autres colonnes sont ajoutées dans le seul but de la génération de tests automatiques par *GreenT*. Voici les informations les plus intéressantes :

Nom de la variable Le nom de la variable testée : il existe un nom court et un nom long.

Informations aidant à la conversion des données Certains équipements¹, à l'instar du *debugger* ne fonctionnent qu'avec des valeurs hexadécimales. À la charge de *GreenT* de convertir ces données vers des valeurs physiques exploitables par le testeur. Ces colonnes contiennent les informations nécessaires au calcul de conversion, informations tel que le domaine de définition physique et le domaine de définition hexadécimal, avec ces deux informations il est possible d'effectuer les conversions physique vers hexadécimal.

¹Vous trouverez plus d'informations sur le fonctionnement d'une table de tests section 1.2.3

Nécessité d'un test automatique Un *GreenTTest* ne sera généré que si la colonne vaut *Yes*.

Statut du test L'outil éditera automatiquement cette colonne afin de reporter le statut du test².

Précondition (cf section 4.3.2) Contient un scénario d'initialisation du *workbench* : tension de départ, démarrage de l'ECU, ...

Scénario de stimulation (cf section 4.3.2) Contient un ou plusieurs scenarii de stimulation destinés à faire générer au HiL un certain nombre de stimuli.

ExpectedBehavior (cf section 4.3.3) Contient une expression évaluant les variables ayant été enregistrées durant la stimulation : *GreenT* devra vérifier que cette expression est correcte à tout instant de la stimulation.

Variable à enregistrer (cf section 4.3.3) Contient les variables devant être enregistrées durant un scénario, en plus des variables présentes dans l'*expected behavior*. Celles-ci peuvent servir en tant que données contextuelles permettant de mieux cerner le résultat d'un test.

Informations du test (cf section 4.3.5) Plusieurs colonnes telles que la sévérité, le responsable du test, des commentaires, ...

1	RB Variable	Test Interface	Test Condition	Test Stimulus	Test ExpectedBehavior	Test RecordedVariable	Test LocalAli	Test Seve
2	ACCIntVlv_iSens	ID	SUB_ECU_GO	SCENARIO ID_VACC_control1 SUB_SCE_BENCH_ACC_0_100_0_POURC_ActiveBenchMode_0 END SCENARIO	EVAL(ACCIntVlv_iSens=(v_pwm_cfb_acc_slv / NC_C			Low
3	ACCIntVlv_r	ID	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO	EVAL(pwm_ducy_acc_slv_cus=ACCIntVlv_r, TOLRES			Low
74	Air_uRawTEGRClrDs	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Air_uRawTEGRClrDs=0)			Low
100	BrkBstP_uRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(BrkBstP_uRaw=0)			Low
117	Clth_b25PrcSens	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_b25PrcSens=0)			Low
133	Clth_uAnaRaw	STUB	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_uAnaRaw=0)			Low

Figure 4.1: Aperçu d'un fichier Walkthrough

4.2 Fonctionnement Global

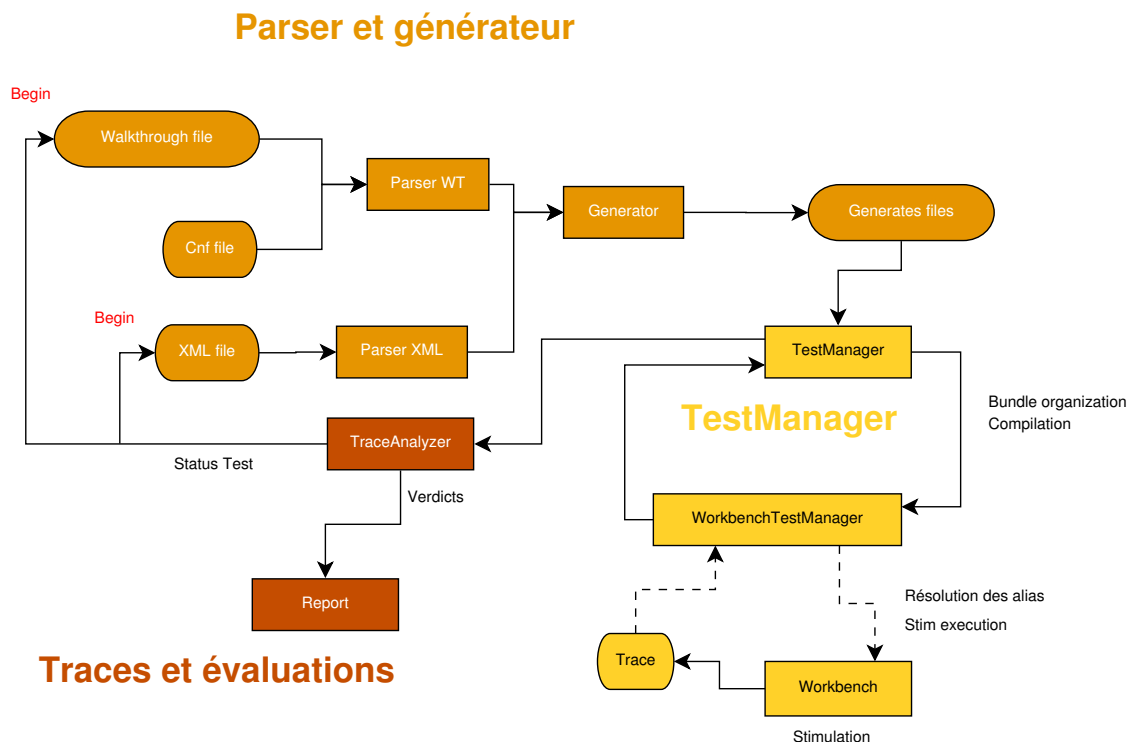
Le développement de *GreenT* inclut un certain nombre de fonctionnalités attendues par le client et indispensable à son fonctionnement. D'autres fonctionnalités pourront apparaître plus tard en fonction des besoins.

Les principaux modules sont les suivants, avec leurs interactions schématisées figure 4.2 : dans des objets ovales sont représentés des fichiers, les carrés représentent des modules de l'outil et les flèches en pointillés un transfert réseau, les couleurs représentent les différents modules de la plateforme.

Dans la figure 4.2, vous pouvez voir des flèches en pointillés représentant des échanges réseau. En effet l'exécution des stimulations et l'enregistrement des traces se fait par un contrôle distant des bancs de tests³.

²Un test pouvant être *Green* ou *Red* mais peut aussi comporter une erreur, tel qu'un problème d'exécution ou de génération, ...

³La schématisation du fonctionnement d'un banc est disponible section 1.2.3 figure 1.5

Figure 4.2: Fonctionnement général de la plateforme *GreenT*

Les principales fonctionnalités demandées par le client sont disponibles sur le client, la majorité peuvent s'effectuer en local :

- Le parsing et la génération (section 4.3.1)
- L'organisation en Bundle afin d'optimiser le temps d'exécution des tests (section 4.3.4)
- La production de rapports détaillés (section 4.3.3)

Alors que d'autres fonctionnalités vont nécessiter la présence de serveurs et de connexion réseau permettant d'effectuer ces actions :

- Les stimulations (section 4.3.2)
- Les enregistrements des traces (section 4.3.2)

4.3 Les fonctionnalités du client

Afin de répondre au mieux aux attentes du client, il a été choisi d'utiliser le langage Java pour développer notre client, ceci en raison de divers avantages comme le côté multi-plateforme, son typage fort et sa forte communauté qui permet ainsi une maintenance facilitée.

4.3.1 Parsing et Génération

Le but premier de la plateforme est d'effectuer des tests automatiques, il est ainsi indispensable d'avoir un système d'automatisation, au travers d'une génération.

Pour cela, nous avons un parser : il analyse un fichier, appelé TestPlan. En debut de stage, nous n'avions que le *Walkthrough* pour débiter, mais ensuite, il a rapidement été nécessaire d'avoir un test plan générique. Notre architecture autorise également l'ajout ultérieur de types de fichiers différents, tel que XML, bases de données, ...

Grâce à ce fichier, on en retire pour chaque test, le scénario de précondition, les différents scénarii de stimulations, leur *Expected Behavior*, les données qui devront être enregistrées ainsi que différentes informations sur le test⁴.

Une fois toutes ces données acquises, il les transmet à un générateur qui est en charge d'écrire les fichiers Java de chaque test, tous sont organisés dans un dossier temporaire avec un dossier par test. Le **TestManager** peut ensuite traiter ces données.

Exemple Le testeur souhaite vérifier le bon fonctionnement des ventilateurs de refroidissement du moteur. Ci-dessous les différentes cellules qui pourraient être renseignées par le testeur pour ce cas de test. Nous verrons ce qu'effectuent précisément ces actions dans la suite de la section.

Précondition	Scénario de Stimulation	Expected Behavior
<pre>// Tension batterie HIL_VB = 13; // Clé moteur HIL_KEY = 1; // Vérifications CHECK(HIL_KEY = 1 && HIL_VB = 13);</pre>	<pre>// Rampe de vitesse véhicule // de 0 à 50km/h par pas de 1 // toutes les secondes RAMP(HIL_VS, 0, 50, 1, 1);</pre>	<pre>if temperature > max then // Ventilateur allumé EVAL(ventilateur_on = 1); else // Ventilateur éteint EVAL(ventilateur_on = 0); end if</pre>

4.3.2 Stimulation

Afin de tester une variable du plugin, les développeurs vont utiliser des alias présents sur un device : actuellement, un HiL ou un debugger, prochainement nous pourrions en utiliser d'autres. Ces alias permettent de simplifier le travail du spécifieur, il n'aura pas à retenir une adresse d'un élément sur le HiL, un simple alias permet de lui faire cette abstraction et ce raccourci.

Le spécifieur va rédiger des scénarii de stimulation, ceci afin de mettre le contrôleur dans certaines conditions. Son but sera ensuite de vérifier que ces variables restent cohérentes vis-à-vis du scénario effectué.

Un scénario particulier doit être spécifié : une précondition qui a pour but d'initialiser les équipements et certains alias afin d'avoir un état de stimulation qui soit cohérent et identique à chaque lancement du scénario. Ce scénario sera effectué avant le lancement de chacun des scénarii de stimulation.

Durant l'exécution d'une stimulation, les variables nécessaires sont enregistrées afin de pouvoir produire des rapports et des verdicts ensuite.

Exemple Comme nous l'avons vu section 4.3.1, le testeur nous a donné un scénario de stimulation ainsi qu'un scénario de précondition. Le code généré va ainsi communiquer avec le serveur HiL afin qu'il envoie les bons stimuli à l'ECU.

Comme mis en commentaires, le scénario de précondition va mettre la batterie à 13 Volts, et mettre la clé, nous allons ensuite vérifier que cette action a bien été faite. Si tel est le cas, le scénario de stimulation va être effectué. Celui-ci va appliquer une rampe de vitesse, notre véhicule va aller de zéro

⁴Responsable du test, sévérité, commentaires, nom de la variable, ...

à cinquante kilomètre-heure avant de retourner à l'arrêt.

4.3.3 Les traces et leurs évaluations

Lorsqu'un scénario de stimulation s'exécute, un certain nombre de variables sont enregistrées : ces variables sont stockées sous la forme d'une trace au format CSV⁵, qui pourra plus tard être représentée sous forme de courbe.

Une fois que la trace est complète, il est nécessaire de l'évaluer : le spécifieur a décrit le comportement attendu dans la colonne *Expected Behavior* détaillant dans quel cas le test est correct. Cette expression va être transformée en arbre logique afin de l'évaluer à tout instant de la trace.

Exemple Durant notre rampe véhicule, le *debugger* a enregistré différentes variables, en l'occurrence nous avons enregistré les trois variables présentes dans notre *Expected Behavior* : la température du moteur – `temperature`, la température maximum sans ventilateur – `max`, ainsi que l'état de notre ventilateur – `ventilateur_on`.

Ces variables ont été enregistrées durant l'intégralité de la rampe véhicule qui à eu une durée de cinquante secondes. À la fin de la stimulation, le serveur retourne ainsi une trace de cinquante secondes contenant toutes les variables. *GreenT* peut ensuite évaluer notre *expected behavior* sur l'intégralité de l'enregistrement.

4.3.4 Le module TestManager

Comme le montre la figure 4.2, la classe `TestManager` est le chef d'orchestre de *GreenT*, il a donc un certain nombre de responsabilités.

Il va d'abord organiser les différents tests en un concept que nous avons appelé *Bundle*, ceci dans un but d'optimisation du temps d'exécution. En effet, si nous avons 1000 tests de 2 minutes, cela ferait plus de trente heures d'exécution. Pour palier à ce problème, nous avons deux stratégies :

- Le regroupement de tests ensemble, si deux tests possèdent le même scénario de stimulation, alors nous n'exécuterons qu'une seule fois ce scénario, et évaluerons la trace pour chacun des tests. Ce regroupement est appelé «Bundle».
- Même avec notre stratégie des Bundles, l'exécution pourrait être encore trop longue. Ainsi, il est possible d'utiliser plusieurs tables en simultané. Le temps d'exécution est ainsi divisé par le nombre de tables.

Afin d'être le plus souple possible, il existe plusieurs modes d'exécution de la classe `TestManager` :

Check only Essaye de parser les différents fichiers, et vérifie que ceux-ci ne comportent aucune erreur de grammaire, d'alias introuvable, d'écriture sur un alias en lecture seule etc... Cela permet à l'auteur des cas de test de se vérifier sans avoir besoin de table de test.

Parse and generate bundles Parse les fichiers et génère des jars exécutables répartis en bundle

⁵Comma Separated Values

Parse and execute Parse les fichiers, génère les jars pour les bundles et les exécute : c'est le mode « classique ».

Restart test execution Redémarre une exécution qui se serait mal terminée. Ceci à l'aide d'une base de données SQLite. Cette base de données contient toutes les informations des différents scénarii et sera capable de redémarrer à l'endroit où une coupure à eu lieu. Cela évite de devoir effectuer de nouveau trente heures d'exécutions si le problème à eu lieu sur la fin.

4.3.5 Production de rapport détaillé

L'outil a en charge la production d'un rapport détaillé pour chaque test. Ce rapport contiendra un certain nombre d'informations, et permettra au testeur de comprendre pourquoi le test n'est pas passé. Voici les informations que contiendra ce rapport :

- Nom du test, de la variable à tester
- Nom du responsable du test
- Sévérité du test
- Branches de l'*Expected Behavior* renvoyant faux(Test « Rouge »), n'ayant pas pu être testé(Test « Gris ») et étant correct(Test Vert)
- Le testeur aura à sa disposition les expressions concernées par un résultat Rouge ou Gris.
- Les colonnes utiles du *Walkthrough*

Actuellement, les rapports se font au format Excel avec l'intégralité de notre enregistrement et pour chaque instant d'enregistrement (timestamp), un verdict. Un exemple de rapport est accessible en Annexe D page 53.

Dans un futur proche, ces rapports pourraient être générés dans un format Web avec une possibilité de naviguer entre plusieurs tests, et d'avoir un affichage des courbes de manière graphique. Une autre possibilité serait de générer des documents Word ou PDF afin de pouvoir facilement transmettre un rapport de test à un client externe à Continental.

4.3.6 Mise à jour du TestPlan

Une fois l'analyse d'un test exécuté, un verdict global est mis dans le fichier TestPlan original : ce verdict est consolidé en fonction des rapports détaillés. Ainsi, un test sera vert si l'expression a été validée sur l'ensemble des traces, le test sera rouge dans le cas contraire.

Remarque En cas d'erreur à la génération^a ou à l'exécution^b, la plateforme doit afficher un message d'erreur clair au niveau du test afin que l'utilisateur soit conscient du problème. Libre à lui de corriger le test si nécessaire, ou de le signaler si cela semble être un bogue. Ce message d'erreur est reporté automatiquement dans le *TestPlan*.

^aMauvaise syntaxe, variables inexistantes, ...

^bProblèmes réseaux, variable non trouvée, communication entre l'ECU et le debugger, ...

4.4 Les fonctionnalités des serveurs

Comme expliqué précédemment, *GreenT* va avoir en charge l'exécution de stimulations. Celles-ci vont communiquer avec une table de test. Actuellement une table est composée de deux équipements différents, comportant chacun leur serveur :

- Un HiL, Hardware In the Loop, simulateur d'environnement véhicule
- Un Debugger permettant de voir l'état du programme présent dans l'ECU

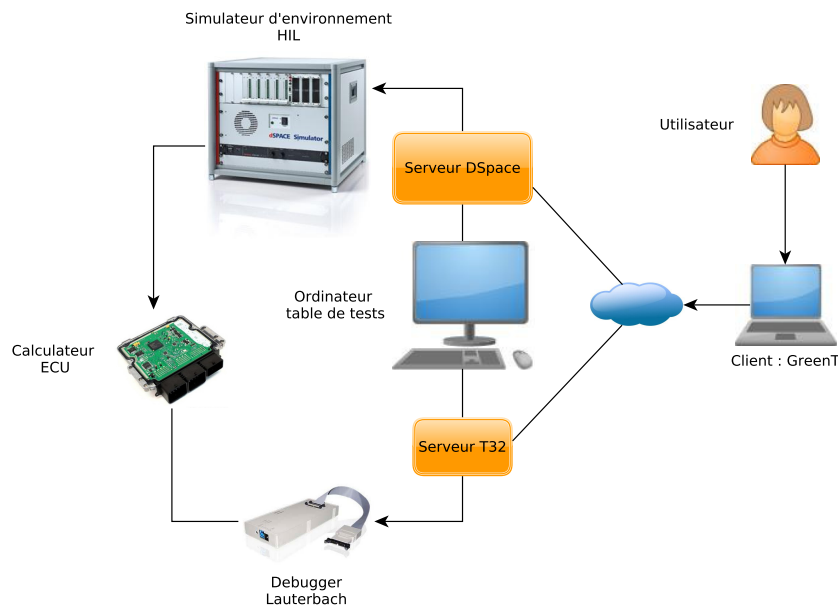


Figure 4.3: Communications de la plateforme

Remarque Un troisième équipement a été ajouté afin de faciliter la synchronisation des traces du HiL et du debugger. L'intérêt de l'analyseur logique est présenté section 5.4.

Au début du développement de la plateforme, il a été décidé que les serveurs devront être le plus simple possible pour plusieurs raisons :

- Donner accès à un maximum de fonctionnalités en mode « offline », c'est-à-dire sans accès à une table de test
- Rabattre le maximum de fonctions métier près du client pour centraliser au maximum le fonctionnement et éviter la maintenance superflue
- Avoir la possibilité de réutiliser les serveurs pour d'autres projets
- Pouvoir ajouter facilement un nouveau *device*, qui ne nécessiterai que l'ajout d'un nouveau serveur relativement simple

4.4.1 Le serveur Debugger, contrôle de Trace32

Le serveur Debugger propose des services basiques permettant de répondre aux besoins :

- Flasher le logiciel dans la flash de l'ECU
- Démarrer l'ECU
- Arrêter l'ECU
- Lire une variable ou une calibration
- Modifier une variable
- Enregistrer des variables



Afin d'effectuer ces actions, le serveur s'appuie sur une API de l'outil Trace32 permettant de contrôler le debugger. Ainsi tous nos services vont s'appuyer sur cette API. Cette API a été développée en interne chez Continental à l'aide de Java Native Access. C'est pour cette raison que ce serveur est développé en

Java afin de pouvoir utiliser facilement ces fonctions.

4.4.2 Le serveur HiL, contrôle du ControlDesk DSpace

Le HiL contient une base de données, représentant un modèle simulant l'environnement véhicule. Ce modèle a pour but de se rapprocher au maximum du fonctionnement réel de notre moteur.

Le serveur DSpace va devoir lui aussi répondre aux différentes stimulations, ainsi ces services sont relativement similaires :

- Modifier une valeur du modèle
- Lire une valeur du modèle
- Enregistrer des valeurs

Ce serveur a été développé en réutilisant une partie de ce qui avait été fait pour la TA3, présentée section 1.2.3 afin de ne pas « réinventer la roue ».

À l'instar du serveur Debugger, nous utilisons une API fournie par l'outil permettant de contrôler le HiL : ControlDesk. C'est ainsi que ce serveur est développé en Python afin de répondre à cette contrainte : l'API du ControlDesk est en Python.



L'utilisation de Thrift, permettant de faire dialoguer du Java côté client et du Python côté serveur facilement, ce sera imposé comme solution.

5

Ma collaboration au projet

Après avoir défini les besoins de notre outil et son fonctionnement général, nous allons maintenant voir en détail de quelle manière j'ai contribué à ce projet. Le développement s'est effectué en deux grandes étapes. D'une part, la finalisation de la première version via la production de rapports détaillés, et ce jusqu'en janvier. Suivi par des améliorations afin que l'outil puisse être utilisé par le plus grand nombre, notamment par les équipes des projets Renault.

Sommaire

5.1	La difficulté de productions de rapports	33
5.2	L'arrivée des projets multi-cœurs . . .	35
5.3	Ouverture de l'outil à d'autres projets	37
5.4	La synchronisation des traces	38

5.1 La difficulté de productions de rapports

La production de rapport est le cœur de notre plateforme. Ces rapports doivent être fiable, afin d'éviter de faire perdre du temps à l'utilisateur en cas de faux rouge, et ne pas manquer de bogue, en cas de faux vert.

Au début du développement de l'outil, une solution avait été mise en place, cependant celle-ci avait été effectuée de façon naïve et ne pouvait fonctionner. C'est ainsi qu'en août, nous avons commencé la conception d'un autre algorithme de production de verdicts, basé sur les récurrence de calculs et le concept d'entrées et de sorties de calculs.

5.1.1 Les calculs du logiciel

Un logiciel embarqué temps réel effectue différents tâches de calculs à des récurrences fixes. C'est-à-dire que le système doit être déterministe et avoir des tâches qui s'exécutent après un temps donné, moyennant une petite marge d'erreur. Dans les logiciels que nous testons, ces tâches peuvent être exécutées toutes les dix millisecondes par exemple.

Une tâche de calcul prends des données en entrée, et écrit un résultat dans une ou plusieurs variables de sortie. Dans le cadre de l'intégration du plugin, un certain nombre de tâche ont pour but la connexion de ce plugin. Ainsi, elles vont prendre les des données du plugin en entrée, et écrire le résultat dans une variable de chez Continental.

5.1.2 Le problème de l'existant

1	EB_EnvT_uRaw	EB_vp_temp_air[2]	EnvT_uRaw	vp_temp_air[2]	[EnvT_uRaw=vp_temp_air[2] * 1000.0, tolRes(1.0)]				
1886	3803.800056681037	3.803710937500004			GREEN				
1887	3803.800056681037	3.803710937500004			GREEN				
1888	3803.800056681037	3.803710937500004			GREEN				
1889	3803.800056681037	3.803710937500004			GREEN				
1890	3803.800056681037	3.774414062500004		0X60A0	RED				
1891	3803.800056681037	3.774414062500004			RED				
1892	3774.400056242943	3.774414062500004	0X49B8		GREEN				
1893	3774.400056242943	3.774414062500004			GREEN				
1894	3774.400056242943	3.774414062500004			GREEN				
1976	3774.400056242943	3.774414062500004			GREEN				
1990	3774.400056242943	3.774414062500004			GREEN				
1991	3774.400056242943	3.774414062500004			GREEN				
1992	3774.400056242943	3.774414062500004			GREEN				
1993	3774.400056242943	3.779296875000004		0X60C0	RED				
1994	3774.400056242943	3.779296875000004			RED				
1995	3779.2000563144684	3.779296875000004	0X49D0		GREEN				
1996	3779.2000563144684	3.779296875000004			GREEN				
1997	3779.2000563144684	3.779296875000004			GREEN				
1998	3779.2000563144684	3.779296875000004			GREEN				
1999	3779.2000563144684	3.779296875000004			GREEN				

Figure 5.1: Exemple d'un rapport incorrect

En début du projet, il avait été décidé que nous allions évaluer une trace à l'ensemble des time-stamps, comme le montre la figure 5.1. Sauf que comme nous pouvons le voir, cette solution ne peut fonctionner. En effet, le *debugger* enregistre tous les changements des variables durant les stimulations.

Or, l'*expected behavior* qui nous est fourni ne peut être vrai à tout instant de la trace, mais doit être vrai à la fin de notre tâche. Pendant l'exécution d'une tâche, ou avant notre tâche, nous pouvons voir des changements sur des variables d'entrée et être dans un état incohérent : ces états ne nous intéressent pas, et ne doivent pas être pris en comptes, sinon, nous allons dire RED sur des instants qui ne sont pas significatifs pour l'utilisateur.

Dans la figure 5.1, nous devons évaluer l'*expected behavior* `EnvT_uRaw=vp_temp_air[2] * 1000`, or nous l'évaluons à tous les instants. Nous prononçons RED à la ligne 1890, car `vp_temp_air[2]` a changé... Mais `EnvT_uRaw` n'a pas eu le temps de s'adapter : il est nécessaire d'attendre sa mise à jour avant de prononcer un réel bogue.

5.1.3 Produire un verdict fiable

La solution qui a été trouvée est d'utiliser le concept de variables d'entrées et de sorties vu précédemment. Nous savons que la tâche de calcul est terminée lorsque la variable de sortie est rafraichie.

Ainsi, un verdict est prononcé uniquement aux rafraichissements de la variable de sortie, entre temps, si les variables d'entrées changent, le verdict est interpolé : nous sommes dans un état intermédiaire. Figure 5.2 montre le rapport du même test que précédemment, avec une prononciation de verdict correcte.

Comme nous pouvons le voir, l'*expected behavior* est la même, mais cette fois-ci, le verdict est prononcé de la bonne manière. Nous mettons « GREEN » lorsque la variable de sortie, `EnvT_uRaw` change de valeur, et dans les autres cas nous interpolons le verdict, en l'occurrence GREEN. D'où le `GREEN_IN`.

2	TimeSta mp	EnvT_uRaw	vp_temp_air[2]	EnvT_uRaw	vp_temp_air[2]	EB Verdict	(EnvT_uRaw= (vp_temp_air[2]*1000.0))	Comments
3		raw	raw	-	V			
52	4,816831		0X5F60	3730,4	3,73	GREEN	GREEN_IN	
53	4,817006	0X48C4		3725,6	3,73	GREEN	GREEN	
54	4,916837		0X5F80	3725,6	3,73	GREEN	GREEN_IN	
55	4,917012	0X48DC		3730,4	3,73	GREEN	GREEN	
56	5,116827		0X5F60	3730,4	3,73	GREEN	GREEN_IN	
57	5,117003	0X48C4		3725,6	3,73	GREEN	GREEN	
58	5,216799		0X5F80	3725,6	3,73	GREEN	GREEN_IN	
59	5,216974	0X48DC		3730,4	3,73	GREEN	GREEN	
60	5,316805		0X5EE0	3730,4	3,71	GREEN	GREEN_IN	
61	5,31698	0X4862		3706	3,71	GREEN	GREEN	
62	5,616778		0X5EC0	3706	3,7	GREEN	GREEN_IN	
63	5,616952	0X484A		3701,2	3,7	GREEN	GREEN	
64	5,71676		0X5E40	3701,2	3,68	GREEN	GREEN_IN	
65	5,716934	0X47E8		3681,6	3,68	GREEN	GREEN	
66	6,016745		0X5E60	3681,6	3,69	GREEN	GREEN_IN	
67	6,01692	0X4801		3686,6	3,69	GREEN	GREEN	
68	6,116769		0X5DC0	3686,6	3,66	GREEN	GREEN_IN	
69	6,116945	0X4787		3662,2	3,66	GREEN	GREEN	
70	6,216976		0X5DA0	3662,2	3,66	GREEN	GREEN_IN	
71	6,217154	0X476E		3657,2	3,66	GREEN	GREEN	
72	6,316816		0X5DC0	3657,2	3,66	GREEN	GREEN_IN	
73	6,31699	0X4787		3662,2	3,66	GREEN	GREEN	

Figure 5.2: Exemple d'un rapport correct

5.2 L'arrivée des projets multi-cœurs

Jusqu'à maintenant, les calculateurs des contrôles moteur fonctionnaient tous en mono-cœur. Ainsi un seul cœur effectuait les opérations, et il n'y avait pas de parallélisation. Ce type de calculateur était utilisé pour la première phase des projets Ford, le Panther Phase 1. Or, le projet GreenT avait pour but premier d'être utilisable pour tous les projets Ford, et notamment le Panther Phase 2. Le panther phase 2 contient beaucoup de variances logicielles ou hardware, ainsi cette base principale est dérivée en plusieurs projets distincts (FPC, FPD, FPE, FPF, ...) en fonction des applications. Tous ces projets utilisent des calculateurs multi-cœur, possédant 3 cœur distincts.

Or, GreenT ayant été initialement conçu pour le Panther Phase 1, il n'avait jamais été notion de multi-cœur. Afin de répondre aux besoins de l'équipe, et dans un but de généralisation de notre outil au plus grand nombre, il a été nécessaire d'étudier l'utilisabilité de l'outil sur des ECU multi-cœur.

5.2.1 Analyse d'impacts

Afin de pouvoir mettre en place notre outil sur les projets multi-cœur, il a d'abord fallu énumérer les actions qui sont faites sur l'ECU par notre plateforme, et ensuite vérifier d'éventuelles différences :

- Flasher un logiciel
- Lire et écrire sur une adresse RAM
- Changer la valeur d'une calibration en flash
- Démarrer le logiciel (CPU GO)
- Arrêter le logiciel

- Enregistrer une trace

Le fonctionnement d'un calculateur multi-cœur

Tout d'abord, afin de pouvoir observer les eventuels modifications à apporter, il faut connaître les spécifications d'un ECU multi-cœur.

Calculs Le principal problème de notre calculateur, celui-ci possède 3 cœurs distincts qui sont indépendants, ils se démarrent ou s'arrêtent indépendamment.

RAM Chaque cœur possède une RAM dite « préférentielle », ces différentes adresses RAM auront des accès plus rapide. Cependant, chaque cœur peut accéder à l'ensemble des adresses RAM. Dans le cadre des projets Ford, toutes les variables nécessaires à nos tests sont sur le même cœur, le cœur 0.

Flash Une seule mémoire flash, ce stockage est indépendant des différents cœurs.

Afin de faire fonctionner notre outil, nous allons devoir :

- Démarrer ou arrêter l'ensemble des cœurs en fonction de l'action qui est souhaitée.
- Lire ou écrire dans les cases RAM depuis le cœur 0.
- La flash est indépendante de l'architecture de notre calculateur.

Remarque Actuellement notre outil n'utilise pas le concept de *breakpoints*. Ce type d'utilisation aurait un impact fort sur notre outil, étant donné que pour du multi-cœur, il serait nécessaire de choisir quel cœur arrêter, ceux-ci étant indépendant. Si ce besoin se fait sentir, il sera nécessaire de prévoir ce cas d'utilisation.

Le fonctionnement du debugger multi-cœur

Le *debugger* Lauterbach multi-cœur est le même que les *debuggers* mono-cœur. La seule différence est la nécessité d'une licence spécifique, actuellement tous les projets possèdent cette licence sur les différents équipements.

Le debuggage sur projet multi-cœur implique d'avoir une fenêtre du *debugger* par cœur, ainsi nous aurons trois fenêtres, respectivement pour les cœurs 0, 1 et 2. L'outil Trace32 possède une fonctionnalité permettant de « synchroniser » les cœurs, ainsi il est possible de le configurer afin que toute les commandes envoyées sur le cœur 0 soit automatiquement envoyée sur les deux autres cœurs.

Cela nous permet notamment de pouvoir démarrer ou arrêter les 3 cœurs au même instant de façon simple.

5.2.2 Les changements de notre outil

Comme nous avons pu le voir section 5.2.1, ce changement majeur pour les projets n'aura que peu d'impact sur notre outil.

Une fois l'analyse d'impact effectuée, il a fallu adapter nos outils pour deux actions :

- Le démarrage et l'arrêt du programme
- Prendre en compte des temps de réactions de Trace32 différents

Le démarrage et l'arrêt du programme

Comme expliqué précédemment, pour démarrer ou arrêter le programme, il est nécessaire de démarrer ou d'arrêter les trois cœurs simultanément. Pour cela, il a suffi qu'au démarrage de Trace32, notre outil configure celui-ci afin de synchroniser l'appel des commandes des trois cœurs. En effectuant un CPU GO, la commande est envoyé automatiquement aux deux autres cœurs.

Les temps de réactions

Le *debugger* possédant trois fenêtres, il a tout de même fallu adapter des temps de réactions qui sont plus longs, notamment pour le démarrage de l'outil, l'arrêt de celui-ci, le flash d'un logiciel, etc... En effet, sans adaptation de ces temps, l'outil renvoyait des exceptions « *timeout* ».

Les autres actions

Nous n'avons eu aucun changement à effectuer pour les autres actions faites par notre outil. En effet, par défaut, toutes nos commandes sont envoyées sur le cœur 0. Or, il est possible de lire ou écrire dans la RAM, et de flasher directement depuis le cœur 0, toutes les variables étant accessibles de n'importe quel cœur.

5.3 Ouverture de l'outil à d'autres projets

Rapidement après la première version multi-cœur de GreenT, le besoin s'est fait sentir d'utiliser GreenT pour certains tests Renault. Or, il n'était actuellement possible de rédiger des tests que via un fichier Walkthrough, avec les bonnes colonnes et les bonnes variables.

Il était donc nécessaire de pouvoir rédiger un test de manière simple, indépendamment du *Walkthrough*. Pour gagner du temps de développement, il a été choisi d'utiliser ici également un fichier Excel, ce qui permet d'avoir un parser qui ressemblera beaucoup au parser original. Ce fichier Excel générique que j'ai mis au point est présenté figure 5.3.

	A	H	I	J	K	L	M	N	O
	Test Name	Branch coverage requirement	Check recurrence (ms)	Pre condition	Stimulation	Expected Behavior	Contextual data	Verdict	GT Message
1	Clth_b25PrcSens		10	SUB_ECU_GO	SCENARIO CALC_MODE_ROAD_VS_50 SUB_SCENARIO_MODE_ROAD_VS_0_50_0 END SCENARIO	EVAL(Clth_b25PrcSens=0) IF (ECBVlv_r < c_pwm_thd_min_egrcb) THEN EVAL(lv_cmd_egr_cool[0]=0, OUT_VAR(lv_cmd_egr_cool[0])) ELSE IF (ECBVlv_r >= c_pwm_thd_max_egrcb) THEN EVAL(lv_cmd_egr_cool[0]=1, OUT_VAR(lv_cmd_egr_cool[0])) ELSE EVAL(lv_cmd_egr_cool[0] = PRE(lv_cmd_egr_cool[0]), OUT_VAR(lv_cmd_egr_cool[0])) ENDIF ENDIF	vb, n	GREEN	->LINK
2	ECBVlv_r	TOTAL	10	SUB_ECU_GO	SCENARIO wait2s SUB_WAIT_2S END SCENARIO			RED	->LINK
3									
4									
5									

Figure 5.3: Exemple d'un test plan générique

Le fichier de testplan générique contient les informations utiles pour bien effectuer notre test, ainsi il est nécessaire de fournir :

- Des informations contextuelles sur le test, tel que le nom, la description, la release concernée, le responsable, l'agrégat
- Des informations utiles à l'analyse, tel que le type de test (Calcul, Stub, IO), la récurrence de calcul, la nécessité de couverture des branches (partielle ou totale), et si le test doit être exécuté
- Les cellules du test (Précondition, Scenarii de stimulation, *Expected Behavior*, Contextual data)
- Les cellules remplies après l'exécution, c'est-à-dire le verdict du test, les messages d'erreurs, liens vers le rapport

Il suffit à l'utilisateur de remplir ce fichier excel, avec une ligne par test qu'il veut faire, de bien renseigner le type de *testplan* dans le fichier de configuration, et ensuite tout le reste du processus sera le même.

Le développement de cette fonctionnalité a nécessité l'adaptation de l'outil à un autre type de parser que le *Walkthrough*, ce que j'ai mis au point avec un simple héritage, ainsi si un jour nous souhaitons un autre type de parser, tel que du XML, ce sera simple à mettre en place.

5.4 La synchronisation des traces

Dans le cadre des projets Ford, l'équipe avait un besoin unique, celui de comparer des variables de l'ECU entre elle, plus précisément des variables du plugin avec des variables du logiciel Continental.

Afin d'ouvrir l'outil au maximum de projets possibles, un autre besoin est apparu : pouvoir comparer une variable ECU avec un symbole venant du HiL. Ce besoin implique d'avoir deux traces d'exécution différentes, une du *debugger*, et une du HiL. Cette fonctionnalité avait été prévu au tout début du projet, ainsi il est déjà possible d'enregistrer une trace provenant de chacun des deux *devices*, mais par manque de temps, nous avons choisis de mettre la synchronisation de côté.

La plus grande difficulté reste dans la synchronisation des temps du HiL et du *debugger*. En effet, entre le moment où on démarre l'enregistrement côté HiL, et que l'on démarre l'enregistrement côté *debugger*, nous allons avoir un décalage de temps pouvant être conséquent. Afin que GreenT soit capable de prononcer des verdicts entre deux traces différentes, il est nécessaire d'avoir la même base de temps et le même timestamp 0.

5.4.1 Trouver deux signaux comparables

Pour pouvoir synchroniser deux traces, il est nécessaire d'avoir un signal identique enregistré sur les deux traces. En fonction de la forme de ce signal, il est ensuite possible de synchroniser nos deux traces, ceci à l'aide des fronts montants et fronts descendants par exemple.

Il est donc nécessaire d'avoir un signal qui soit envoyé par le HiL, vers l'ECU. Ce signal sera ensuite relu côté HiL et lu côté ECU, nous aurons alors deux signaux identiques des deux côtés, moyennant le temps de propagation du matériel, et éventuellement du bruit en fonction de la qualité du signal.

Afin d'être totalement générique, il est nécessaire de trouver un signal qui soit présent sur l'ensemble des projets, afin de ne pas avoir une solution spécifique à un type de projet (Comme pourrait l'être le signal d'embrayage absent des projets de boîte automatique par exemple). Il existe deux types de signaux que nous allons détailler plus bas.

5.4.2 Utiliser un signal analogique : la batterie

Dans un premier temps, nous avons pensé à un signal qui soit indispensable à l'ensemble des projets, et qui soit facile à contrôler, en ayant aucune incidence sur le calculateur, afin d'éviter des effets de bord.

Le signal batterie nous a ainsi apparu évident, il nous suffisait d'effectuer un *glitch* sur la batterie avec un certain pattern, en début et en fin de scénario. Ce signal serait enregistré côté HiL et côté ECU.

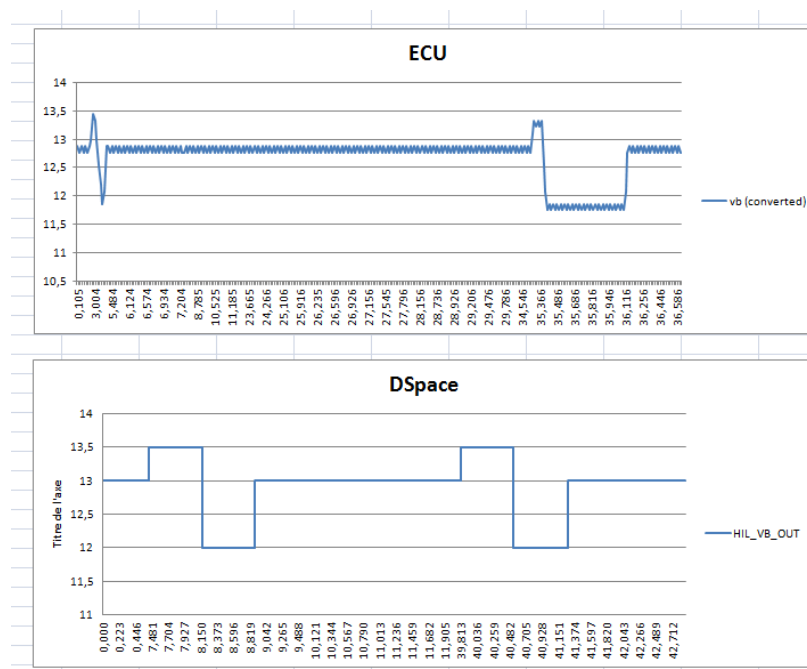


Figure 5.4: Enregistrement de la batterie côté HiL et côté ECU

Comme nous pouvons le voir, le signal de batterie envoyé par le HiL est particulièrement propre, ce qui est normal étant donné que c'est lui qui le génère. Le principal problème vient dans le transport de ce signal jusqu'à l'ECU. Nous pouvons voir que celui-ci est d'une part particulièrement bruité, et d'autre part qu'il est plus faible qu'au départ. Ces problèmes sont matériel, et proviennent principalement du boîtier d'éclatement qui est présent en sortie de HiL, afin de laisser la possibilité à certains utilisateurs de relire différents signaux à l'aide d'un oscilloscope par exemple.

Au vu de ces traces, il est difficile de pouvoir faire correspondre facilement les signaux, en étant certains de ne pas avoir d'erreurs dues au bruit.

Il semble donc bien plus simple d'utiliser un signal logique, ceux-ci ne peuvent être bruités. Il est

ainsi facile de comparer deux signaux logiques via leurs fronts montant ou descendants.

5.4.3 Utiliser le signal logique : le signal clé

Comme vu précédemment, le signal logique est bien plus simple à traiter et limite les erreurs dues au bruit. Il est donc nécessaire de trouver un signal logique qui est disponible sur l'ensemble des projets, et qui n'a aucun impact sur le code du calculateur. En effet, notre signal permettant la synchronisation ne doit pas « polluer » notre test, afin d'avoir des pré-conditions correctement établies.

Nous n'avons pas pu trouver un signal correspondant exactement à ces critères, nous avons pensé à la pédale de frein ou d'embrayage, utilisées à l'arrêt, mais d'une part nous ne pouvons garantir l'absence de stratégies en fonction des projets, et d'autre part une voiture automatique ne possèdera pas d'embrayage. Une autre solution a été trouvée : la clé.

La clé possède un certain nombre de stratégies (effacer des erreurs par exemple), cependant il n'est pas possible d'avoir un test où la clé n'est pas mise à 1, le calculateur n'étant pas alimenté sans clé, aucune trace ne pourrait être enregistrée. De la même manière, un test doit obligatoirement terminer sur une clé à 0, ainsi tous les tests effectués avec GreenT possèderaient les étapes montrées figure 5.5.

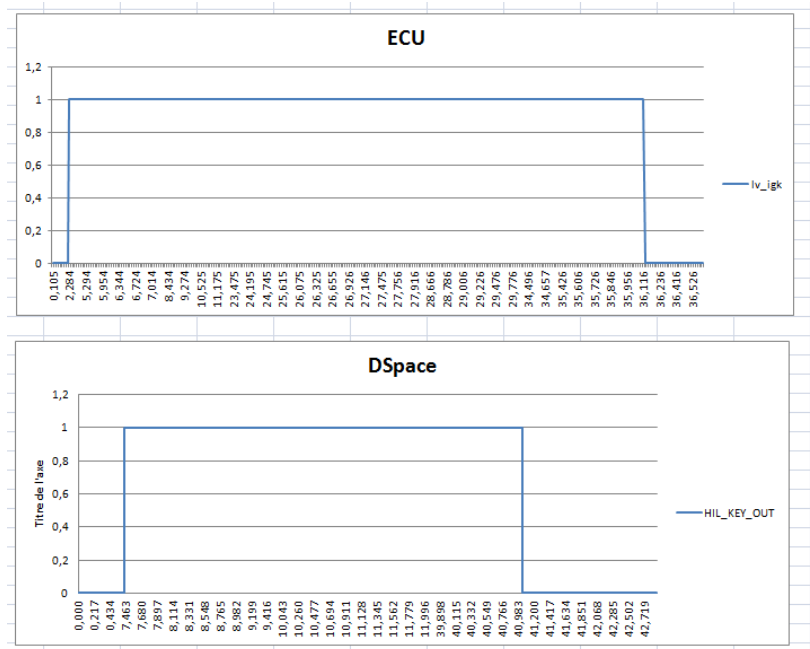


Figure 5.5: Enregistrement idéal d'un signal logique, ici la clé

Tous les tests possédant ces fronts montants et descendant, il est ainsi possible de synchroniser nos deux traces en enregistrant la clé côté HiL, et côté *debugger*.

Seulement, comme nous allons le voir, la figure 5.5 montre l'enregistrement idéal. Malheureusement il n'est pas possible d'enregistrer la clé côté ECU de cette manière.

5.4.4 Le problème de l'enregistrement de la clé

Le signal clé semble la solution idéale de synchronisation. Cependant, ce signal est particulier étant donné que c'est sa mise à 1 qui permet d'alimenter le calculateur. Il est donc impossible d'enregistrer un front montant de la clé via le *debugger*, celui-ci fonctionnant avec le port JTag, il est nécessaire que le calculateur soit alimenté.

Les traces d'enregistrement

La solution trouvée est d'enregistrer le signal de clé avec un autre équipement fourni par Lauterbach, l'entreprise délivrant le *debugger*. Cet outil est un analyseur logique, appelé *PowerProbe*, qui se branche directement sur le *debugger* via un port série. Il est ainsi possible d'enregistrer le signal clé via un fil allant du HiL jusqu'à l'analyseur logique.

Nous aurons donc trois traces différentes pour la synchronisation :

- La trace venant du HiL, contenant le signal clé
- La trace venant du *PowerProbe*, contenant le signal clé et une pulse de synchronisation
- La trace venant du *debugger*

Le *PowerProbe* étant branché en série sur le *debugger*, il est possible de paramétrer le *debugger* afin d'envoyer une pulse au *PowerProbe* dès que l'ECU est démarré, il suffira ensuite de synchroniser le 0 de la trace *debugger* avec la pulse du *PowerProbe*. Restera à synchroniser la trace du *PowerProbe* avec la trace du HiL en fonction des fronts du signal clé. Figure 5.6 est représenté les trois traces qui seront synchronisées.

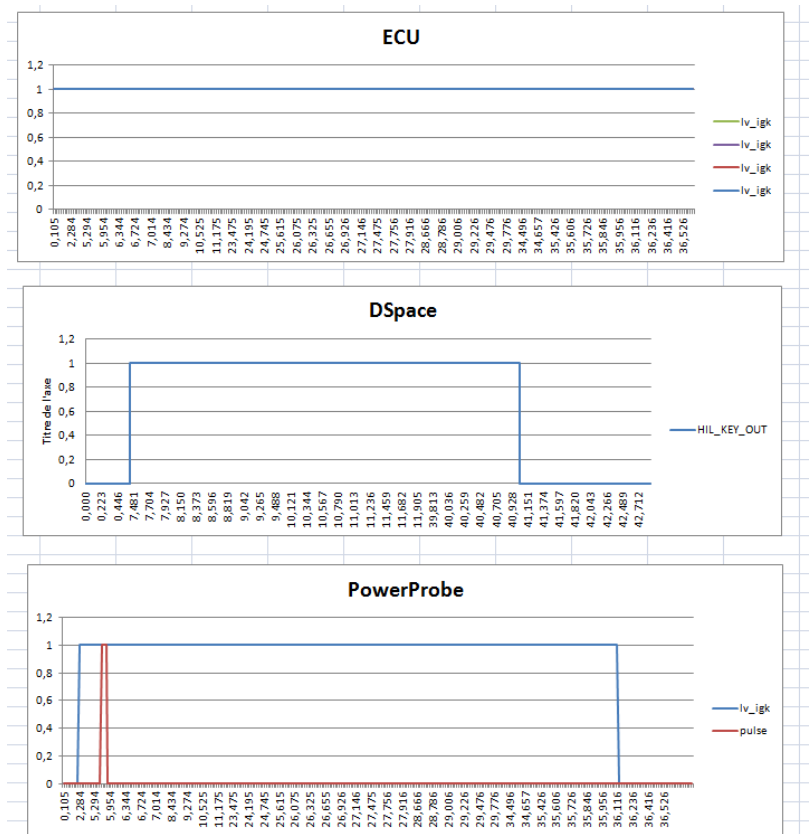


Figure 5.6: Enregistrement de la clé sur les trois équipements

Sur la figure 5.6, nous pouvons observer d’une part l’enregistrement du HiL, qui n’a pas changé, l’enregistrement réel côté ECU qui n’est pas utile, car il n’est capable de ne voir que la clé à 1, et surtout l’enregistrement du *PowerProbe*. On peut voir sur celui-ci la clé, en bleu, ainsi qu’une pulse en rouge correspondant à l’instant où l’ECU a démarré.

La configuration permettant l’enregistrement

Afin d’avoir les traces comme présentées section 5.4.4, il est nécessaire de brancher correctement les équipements sur la table de tests. Cette configuration est légèrement différentes que lorsque nous n’avons pas de synchronisation, celle-ci est détaillé dans la figure 5.7.

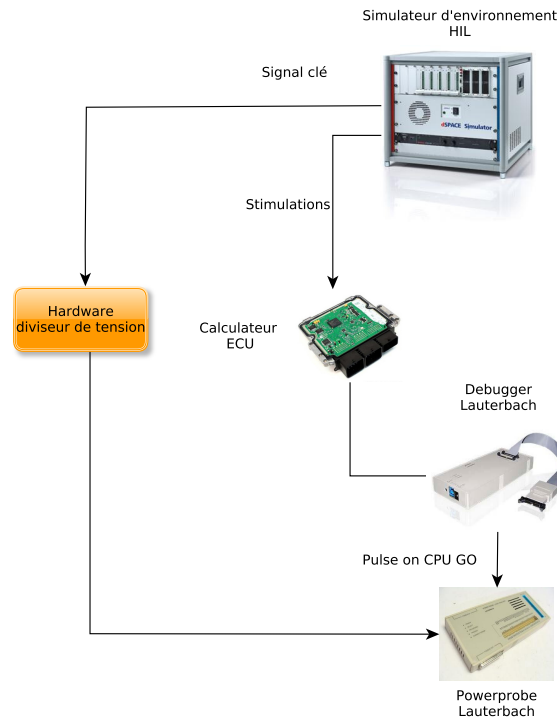


Figure 5.7: Configuration des équipements de la table de tests

Le *PowerProbe* se branche simplement en série sur le *debugger*, un port est prévu à cet effet, il faut ensuite configurer correctement le *debugger* via des commandes spécifiques permettant d’envoyer une pulse au démarrage de l’ECU.

Le HiL permet de relire les valeurs en sortie du dspace, il est ainsi possible de brancher un fil au niveau de la sortie du signal clé, et de le relier à une entrée du *PowerProbe*. Celui-ci enregistrera à échantillonnage fixe le signal sur l’entrée correspondante.

Remarque Il a été nécessaire d’avoir un petit étage *hardware* entre le HiL et le *PowerProbe*. En effet, le signal clé d’une voiture est en 13 Volts, et peut monter jusqu’à plus de 20 volts pour un camion. Or notre équipement *PowerProbe* ne peut accepter des voltages aussi conséquents, une équipe au sein de Continental nous a fabriqué un équipement permettant de sortir une tension bien plus faible afin d’éviter d’endommager le *PowerProbe*. Cet étage est branché entre le HiL et le *PowerProbe*.

À terme, il sera nécessaire d’avoir plusieurs de ces adaptateurs *hardware* afin de les fournir aux différentes équipes de tests, ceux-ci devront pouvoir les brancher facilement.

Après ce stage d'un an, il est temps de dresser un bilan, du point de vue de Continental afin de voir ce que mon travail leur a apporté, mais également en quoi cette alternance d'un an a été bénéfique pour ma future carrière professionnelle.

Sommaire

6.1	Bilan pour Continental	43
6.2	Bilan personnel	44

6.1 Bilan pour Continental

Mon travail dans l'équipe de développement aura été intéressant pour l'entreprise, en partie grâce à ma connaissance de l'outil suite à mes stages précédents. En effet, avoir contribué au projet les années précédentes sur la conception et le développement de celui-ci m'a permis de d'être très rapidement opérationnel, et notamment de revenir en septembre directement là où je m'étais arrêté, c'est-à-dire la conception du nouveau système de verdicts.

Nous avons pu livrer une première version de l'outil en janvier après deux ans de développement. Cette première version nous a permis de mettre en place le cœur de l'outil sur lequel j'ai pu ajouter de nouvelles fonctionnalités, et construire autour d'une base solide.

Comme présenté chapitre 5, j'ai très rapidement mis en place l'adaptation de GreenT aux projets multi-cœurs, cette adaptation a permis à l'équipe de développement des projets Ford Panther Phase 2 d'utiliser l'outil. C'est ainsi que cette équipe a été capable de rédiger 270 tests, avec peu de support, principalement l'utilisation de la documentation, ces tests peuvent ensuite être exécutés sur table de tests en 2h30. L'équipe peut donc effectuer des tests systématiques à l'ensemble des versions logicielles de leur projets de manière rapide, et grâce à l'outil il a déjà été possible de trouver une dizaine de bogues.

Le projet n'est pas encore terminé, en effet il reste encore beaucoup de fonctionnalités utiles, du support et de la maintenance à effectuer, je vais notamment continuer le développement de la synchronisation aux mois de Juillet et Août, tout en effectuant la transmission de connaissance. En effet, dès septembre, Alain FERNANDEZ sera en charge de maintenir le projet et devra donc avoir toutes les cartes en mains pour être le plus efficace possible.

De plus, grâce à la synchronisation, on peut espérer que l'outil sera ensuite utilisé sur les projets Renault, ce qui leur fera gagner du temps, mais demandera plus de support.

6.2 Bilan personnel

Cette expérience en entreprise m’a beaucoup apporté, tout d’abord d’un point de vue technique, j’ai acquis de l’expérience en conception logicielle, grâce à toutes nos réunions où nous réfléchissions à la meilleure approche possible. De plus lors de problèmes, le travail en équipe m’a permis d’avoir une autre vision du problème et une autre manière de le résoudre !

J’ai également approfondi le domaine de l’automobile, notamment en étudiant le fonctionnement des ECU multi-cœur ou du fonctionnement du debugger avec l’utilisation du port JTag. J’ai pu aussi apprendre à utiliser l’analyseur logique de Lauterbach, le *powerprobe*, et des notions d’électronique m’ont été utiles afin de ne pas endommager le matériel. J’aurai ainsi pu voir que l’informatique ne peut se suffire à elle même et nécessite la connaissance d’un domaine métier.

J’ai pu mettre en pratique les enseignements du Master DL, notamment ceux de M1 avec les modules MCPOO pour la conception UML, DCLL pour l’utilisation de Git, mais également le cours d’IVVQ nous ayant montrés les approches de validation et de vérification, celui de PCR pour les concepts que j’ai utilisés avec Thrift et enfin le cours d’IS pour la partie sur les exigences.

¹. Pour certaines parties du projet, j’ai appris en autodidactie, notamment l’utilisation de Python, ce projet est donc particulièrement intéressant par le nombre de technologies et savoirs dont il a nécessité.

Mais j’ai aussi acquis des connaissances humaines avec notamment le travail en équipe, communiquer sur nos avancements, de manière écrite ou orale, et être capable de synthétiser ses propositions de manière claire et concise. Étant le seul développeur de nouvelles fonctionnalités depuis janvier, j’ai également pu gagner en expérience sur l’analyse des besoins et la synthétisation de ceux-ci.

J’ai également eu le plaisir de revenir dans une multinationale, avec des collègues souhaitant toujours transmettre leurs connaissances et leur expérience, notamment dans le monde de l’automobile et de l’embarqué.

Un bilan très positif donc, qui m’a réconforté dans mon projet professionnel : le développement logiciel, d’outil ou de client lourd dans l’industrie, et de me diriger vers de l’expertise technique. Afin de pouvoir suivre ce projet, j’ai choisi d’aller travailler dans une société de service, Extia, qui me permettra ensuite de travailler au sein d’un client intéressant.

¹Plus de détails sur les différents modules sont disponible dans les références annexe C

Annexes

Ci-après vous trouverez un certain nombre d'annexes qui pourront vous permettre de mieux comprendre l'étendue de mon travail et de la plateforme qui est en cours de développement.

Vous y trouverez ainsi un glossaire, des références, des exemples de rapport, de fichiers générés ainsi qu'une explication plus approfondie des équipements utilisés.

A

Acronymes et Glossaire

API Application Programming Interface, ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Analyseur logique L'analyseur logique est un outil de mesure permettant de connaître au fil du temps l'évolution binaire des signaux (0 et 1) sur plusieurs voies logiques : bus de données, entrées-sorties d'un microcontrôleur ou d'un microprocesseur.

Antlr *Another Tool for Language Recognition*, outil permettant de faciliter l'interprétation d'une chaîne de caractère, celui-ci prend en entrée une grammaire, et génère un arbre syntaxique dans plusieurs langages.

Calibration Valeur stockée en flash pouvant contenir une information permettant de simplifier la configuration véhicule. Une calibration pourrait être le nombre d'injecteurs.

CAN *Controller Area Network*, un bus système série très répandu dans beaucoup d'industries, notamment l'automobile. Ce bus permet de raccorder à un même bus un grand nombre de calculateurs qui communiqueront à tour de rôle.

ControlDesk Outil permettant de piloter le HIL, l'interface permet ainsi de modifier des valeurs de l'environnement véhicule, ou de pouvoir les lire graphiquement.

Device Les différents équipements dont pourrait avoir besoin l'utilisateur : Hil, Debugger, ...

DSPACE Société allemande fournissant Continental en simulateur d'environnement HiL, associé à une interface ControlDesk.

ECU Electronic Control Unit, calculateur du contrôle moteur.

Excel Logiciel tableur appartenant à la suite de Microsoft Office®. Il est possible de modifier une feuille de calcul depuis un logiciel ou un script, notamment en Java.

Flash La mémoire flash est une mémoire de masse non volatile et réinscriptible. Ainsi les données sont conservées même si l'alimentation est coupée.

Flasher Action d'écrire sur la flash, dans notre cas il s'agit d'écrire ou de mettre à jour le logiciel présent sur la mémoire flash de l'ECU.

Grammaire Formalisme permettant de définir une syntaxe claire et non ambiguë.

HiL *Hardware in the loop*, permet de simuler un environnement véhicule autour du calculateur du contrôleur moteur : celui-ci réagira comme s'il était embarqué dans une voiture.

JAR Java ARchive est un fichier ZIP utilisé pour distribuer un ensemble de classes Java.

Java Langage de programmation orienté Objet soutenu par Oracle. Les exécutable Java fonctionnent sur une machine virtuelle Java et permettent d'avoir un code qui soit portable peut importe l'hôte.

JSON JavaScript Object Notation est un format de données textuelles, générique, dérivé de la notation du langage JavaScript, il permet de représenter de l'information structurée.

JVM *Java Virtual Machine*.

Lauterbach Société allemande spécialisée dans les outils de debuggage de systèmes embarqués. Nous utilisons leur debugger, ainsi que leur analyseur logique.

Logiciel de versionnement Logiciel, tel que *Git*, permettant de maintenir facilement toutes les versions d'un logiciel, mais aussi facilitant le travail collaboratif.

Parsing Processus d'analyser de chaîne de caractère, en supposant que la chaîne respecte un certain formalisme.

PowerProbe Analyseur logique de la suite Trace32 développé par la société Lauterbach.

Python Langage interprété de programmation multi-plateforme et multi-paradigme. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire et d'un système de gestion d'exceptions.

Release Version d'un logiciel qui correspond à un état donné de l'évolution d'un produit logiciel utilisant le versionnage. Ainsi, chez Continental, un projet comporte une multitude de versions différentes.

TestPlan Document détaillant les objectifs et la manière de tester une version d'un logiciel sous tests.

Apache Thrift Langage de définition d'interface conçu pour la création et la définition de services pour de nombreux langages. Il est ainsi possible de faire communiquer deux problèmes dans deux langages différents : Python et Java dans notre cas.

Trace32 Logiciel permettant de communiquer avec le debugger JTag, ainsi qu'avec les équipements branchés en série, notamment le PowerProbe.

UML Unified Modeling Language est un langage de modélisation graphique. Il est utilisé en développement logiciel et en conception orienté Objet afin de représenter facilement un problème et sa solution.

XML Extensible Markup Language est un langage de balisage générique permettant de stocker des données textuelles sous forme d'information structurée.

B

Index

TestPlan, 25

Voici les références que j'ai utilisées durant ce stage : particulièrement de la documentation, quelques livres dans lesquels j'ai lu les chapitres qui m'intéressaient pour résoudre un certain problème, certains cours enseignés durant mon cursus, et enfin des sites web et forums me permettant de résoudre des problèmes spécifiques.

C.1 Documentations

C.1.1 En ligne

Documentation Thrift <https://thrift.apache.org/docs/>

Documentation Antlr <http://www.antlr.org/api/>

Documentation Freemarker <http://freemarker.org/docs/>

Documentation Git <http://git-scm.com/documentation>

Documentation Java <http://docs.oracle.com/javase/6/docs/api/>

Documentation Python <https://docs.python.org/2.6/>

C.1.2 Chez Continental

Documentation Lauterbach Documentations sur le fonctionnement du debugger lauterbach, du langage de scripts ainsi que de l'analyseur logique.

Documentation ASAP2 Documentation sur la norme ASAP2, notamment utilisée pour la rédaction de fichier A2L.

Documentation Infineon Documentation permettant de comprendre globalement le fonctionnement du calculateur, et permettant de comprendre des problèmes posés par le debugger Lauterbach.

C.2 Livres

UML2 par la pratique – Étude de cas et exercices corrigés sixième édition, 2008. Rédigé par Pascal ROQUES

Design Patterns : Elements of Reusable Object-Oriented Software Second edition, 1999.
Rédigé par le Gang of four : Erich GAMMA, Richar HELM, Ralph JOHNSON et John VLISSIDES.

The definitive Antlr4 reference 2013, Rédigé par Terence PARR

The L^AT_EX Companion 2nd édition, 2004. Rédigé par Franck MITTELBAACH, Michel GOOSSENS, Johannes BRAAMS, David CARLISLE et Chris ROWLEY

C.3 Cours magistraux

Programmation concurrente et répartie 2016, M2 Informatique Développement Logiciel, Jean-Paul ARCANGELI

Ingénierie Système 2015, M2 Informatique Développement Logiciel, Henri CUILLER

Intégration Vérification Validation Qualification 2015, M2 Informatique Développement Logiciel, Franck SYLVESTRE

Design patterns 2015, M1 Informatique Développement Logiciel, Jean-Paul ARCANGELI

Management de Projets Informatiques 2015, M1 Informatique Développement Logiciel, Bernard CHERBONNEAU

Modélisation Conception Programmation Orienté Objet 2014, M1 Informatique, Ilena OBER

Traduction de langages 2015, M1 Informatique, Christine MAUREL

Langages et automates 2013, L3 Informatique, Christine MAUREL et Jean-Paul ARCANGELI

Construction et réutilisation de composants logiciels 2014, L3 Informatique parcours ISI, Christelle CHAUDET

Conception UML 2012, DUT Informatique, Thierry MILLAN

C.4 Sites Web et forums

StackOverflow <http://stackoverflow.com>

Developepez <http://www.developepez.com/>

Wikipédia <http://en.wikipedia.org/wiki/>

D

Exemple de rapport généré par GreenT

GT_ENSS_Strt_stPs report				
Responsible	O. Jauffrit			
Agregate	ENSS			
Summary	The output of the starter to the powerstage			
Coverage required	TOTAL			
Test type	IO			
Precond-stimulation	SUB_ECU_GO			
Stimulation scenario	SCENARIO start_engine_stop SUB_SCE_START_ENGINE_STOP END SCENARIO			
Reccurrence	5 ms			
Expected Behavior	IF (n <= c_n_max_thd_str_clc_act) THEN EVAL(Strt_stPs=lv_ena_str, OUT_VAR(Strt_stPs)) ELSE EVAL(Strt_stPs=PRE(Strt_stPs), OUT_VAR(S			
Test's verdict	RED			
Eval clauses	EvalClause	Statut	Output variables	Input variables
	((n <= c_n_max_thd_str_clc_act) -> (Strt_stPs = lv_ena_str))	GREEN	Strt_stPs	lv_ena_str, c_n_max_thd_str_clc_act, n
	((NOT((n <= c_n_max_thd_str_clc_act)) -> (Strt_stPs = PRE(Strt_stPs))))	GREY	Strt_stPs	c_n_max_thd_str_clc_act, n
Software under tests	FPD100			
GreenT	1.1.0			
DSPACE server	1.1.0			
DBG server	1.2.0			
Execution date	2016-04-25			
Logs				

Figure D.1: Exemple d'un extrait de rapport généré par *GreenT* – Onglet *Summary*


GT_ENSS_Strt_stPs with trace DbgTrace_Bundle_62_Anonymous_StimScenario_0_start_engine_stop42_1.csv														
Timestamp	lv_ena_str	c_n_max_thd_str_clc_act	lv_str_act_req	n	Strt_stPs	lv_ena_str	Strt_stPs	c_n_max_thd_str_clc_act	lv_str_act_req	n	EB Verdict	((n<=>[Strt_stPs=lv_ena_str])>[Strt_stPs=lv_ena_str])	((NOT((n<=>[Strt_stPs=lv_ena_str])>[Strt_stPs=lv_ena_str]))&&[Strt_stPs=lv_ena_str])	Comments
	raw	raw	raw	raw	raw	-	-	rpm	-	rpm				
	0 0XEE													
0,00012		0X0514				238,00	0,00	0,00	0,00	0,00	NA	NA	NA	
0,000159			0XD0			238,00	0,00	1300,00	0,00	0,00	NA	NA	NA	
0,000311				0X4547		238,00	0,00	1300,00	208,00	0,00	NA	NA	NA	
0,000388					0X9C	238,00	156,00	1300,00	208,00	17735,00	NA	NA	NA	
0,000799					0XD0	238,00	0,00	1300,00	208,00	17735,00	NA	NA	NA	
0,001146				0X0000		238,00	0,00	1300,00	208,00	0,00	NA	NA	NA	
0,001263	0XD0					0,00	0,00	1300,00	208,00	0,00	NA	NA	NA	
0,001412			0XD0			0,00	0,00	1300,00	0,00	0,00	NA	NA	NA	
3,170523			0XD1			0,00	0,00	1300,00	1,00	0,00	GREY	GREEN	GREY	
3,216163					0XD1	0,00	1,00	1300,00	1,00	0,00	GREY	GREEN_IN	GREY_IN	
3,216211	0XD1					1,00	1,00	1300,00	1,00	0,00	GREY	GREEN	GREY_IN	
3,479546				0XD024		1,00	1,00	1300,00	1,00	36,00	GREY	GREEN_IN	GREY_IN	
3,516096				0XD025		1,00	1,00	1300,00	1,00	37,00	GREY	GREEN_IN	GREY_IN	
4,589096				0XD020		1,00	1,00	1300,00	1,00	32,00	GREY	GREEN_IN	GREY_IN	
4,680138			0XD0			1,00	1,00	1300,00	0,00	32,00	GREY	GREEN_IN	GREY_IN	
4,680361					0XD0	1,00	0,00	1300,00	0,00	32,00	GREY	GREEN_IN	GREY_IN	
4,680404	0XD0					0,00	0,00	1300,00	0,00	32,00	GREY	GREEN	GREY_IN	
5,295908				0XD000		0,00	0,00	1300,00	0,00	0,00	GREY	GREEN_IN	GREY_IN	

Figure D.2: Exemple d'un extrait de rapport généré par *GreenT* – Onglet *trace*

Ce rapport est un rapport vérifiant que la variable `EnvT_tSens=temp_air_mes[2]`, moyennant une tolérance.

- Dans la colonne A, nous avons le timestamp, le temps durant lequel la valeur a été enregistrée

- Dans les colonnes B et C, nous avons la valeur de nos variables, en physique, c'est-à-dire converties par *GreenT*
- Dans les colonnes D et E, c'est la valeur hexadécimal, c'est-à-dire la valeur enregistrée. Cette valeur est reportée dans le rapport uniquement au moment où elle a changé. ça peut permettre à l'utilisateur de connaître les instants où une variable est modifiée.
- Et enfin, dans la colonne F, le résultat de l'évaluation pour chaque instant de la trace; En l'occurrence, il est toujours *GREEN*.



Delivery Note

P E S S E A V T A S

Tool: GreenT	Tool Version: 1.1.0	Delivery Date: 04/05/2016
---------------------	----------------------------	----------------------------------

Delivery Type:

<input checked="" type="checkbox"/> Tool Release	<input type="checkbox"/> Patch Release (bugfix only)	<input type="checkbox"/> Prototype
<input type="checkbox"/> Platform Release	<input type="checkbox"/> Patch Release (functional update only)	<input type="checkbox"/> Pilot baseline

Delivery Locations:

<input type="checkbox"/> all TM locations	<input type="checkbox"/> CHG	<input type="checkbox"/> RBG	<input type="checkbox"/> SBA
<input type="checkbox"/> ABH	<input type="checkbox"/> IAS	<input checked="" type="checkbox"/> TLS	<input type="checkbox"/> SHA
<input type="checkbox"/> BLR - IDC	<input type="checkbox"/> ICH	<input type="checkbox"/> TSR	
<input type="checkbox"/> BLR – SISL			

Tool description and Known Issues

GreenT is a tool for test automation on ES project using laboratory workbench equipped with HIL DSpace and Debugger T32 Lauterbach.

GreenT was designed and develop in order to allow ES customer project team to:

- Write themselves their own test case with a simple language (GTL standing for GreenT Language)
- Execute automatically the test using remote access to workbench computer.
- Analyse easily test result to find quickly most meaningful ones (i.e red status) among a large amount of test report

To perform test execution on the workbench, GreenT need to run 3 separated process:

- A GreenT client (typically ran on the user's computer)
- A T32 Debugger server and a HIL Dspace server which must be both executed on the workbench computer

Added features

- Management of multi-core projects
- Parse .gri files for NC_constants
- Add a parameter of time tolerance in testsession : if eb is false, we try it few micros later.
- Add a Generic Testplan for projects that don't use the Walkthrough file
- Scheduling an execution at a given hour
- Improved the writing of sub scenario.
- Add bitwise operators : AND, OR, NOT and XOR in expected behavior
- Migration of GreenT client to Java 8
- Corrections of bugs

Contained Deliverables (e.g. tool, documentation, etc) :

Name	Version	Comment/link
GreenT_1.1.0/	1.1.0	Folder containing Java binary of GreenT client
DBGServerT32_1.2/	1.2	Folder containing executable jar Debugger server on the workbench computer, and all used .cmm script and dll libraries used by the T32 server
HILServerDSpace_1.1/	1.1	Folder containing DspaceServer python script, as well as all scripts used to run the Dspace server
GreenTUserManual_1_1.pdf	1.1	User manual
launch-servers.bat		To launch execution of the T32 and ContolDesk servers on the workbench computer
GreenT_1.1.0.bat	1.1.0	Script for GreenT launching : automatically run the jar version with an embedded JRE1.8.

GreenT 1.1.0 requires following components :

Name	Version	Comment/link
DBGServerT32	1.2 or higher	
HILServerDSpace	1.1 or higher	
Java Runtime Environment (JRE)	1.8.0_51	For all pc without Java 8.

Filename: GreenT_v1.1.0_DeliveryNote.pdf	Page 1 of 2	© Continental Automotive AG For internal use only.
--	----------------	---

Figure E.1: Release note de la versions 1.1.0

F

Exemples de fichiers générés

F.1 Exemple de StimScenario

```
package GT_ECM2_MoFInjDat_phi.stim;

import java.util.List;
import org.apache.log4j.Logger;
import com.continental.gt.devices.Device;
import com.continental.gt.util.UtilExceptions;

import com.continental.gt.devices.Hil;
import com.continental.gt.test.stim.StimScenario;
import java.lang.InterruptedException;
import org.apache.thrift.TException;
import com.continental.gt.test.alias.writable.AliasHilParam;

/**
 * Test of stubbed class generated
 * Generated by GreenT
 */
public class StimScenario_0_CALC_MODE_ROAD_VS_20041 extends StimScenario {
    private Logger log = ↵
        Logger.getLogger(StimScenario_0_CALC_MODE_ROAD_VS_20041.class);

    private Hil hil;

    public StimScenario_0_CALC_MODE_ROAD_VS_20041(){
        super("Anonymous StimScenario_0_CALC_MODE_ROAD_VS_20041");
    }

    public StimScenario_0_CALC_MODE_ROAD_VS_20041(String name) {
        super(name);
    }

    @Override
    public void addAllRequiredAlias() {

        addAliasWritable(Hil.class, "ROAD_MODE_VS_REGUL");
        addAliasWritable(Hil.class, "HIL_VS_SP");
        addAliasWritable(Hil.class, "HIL_KEY");
        addAliasWritable(Hil.class, "PLA_C_KEY_PSN");
    }
    /**
     * @see com.continental.gt.test.stim.StimScenario#exec()
     * Generated by GreenT.
     */
    @Override
```

```

public void exec(List<Device> devices) throws TException, ←
    InterruptedException {

    showMsg(".exec() : executing stimulation code of ←
        StimScenario_0_CALC_MODE_ROAD_VS_20041 class...");
    hil = (Hil)getDeviceByClass(devices, Hil.class);

    ((AliasHilParam)hil.getAliasWritableByName("HIL_KEY")).setPhy((double) 1, ←
        hil);
    ((AliasHilParam)hil.getAliasWritableByName("ROAD_MODE_VS_REGUL")).setPhy((double) ←
        0, hil);
    ((AliasHilParam)hil.getAliasWritableByName("PLA_C_KEY_PSN")).setPhy((double) ←
        1, hil);
    delay(5);
    ((AliasHilParam)hil.getAliasWritableByName("PLA_C_KEY_PSN")).setPhy((double) ←
        0, hil);
    delay(1);
    ((AliasHilParam)hil.getAliasWritableByName("HIL_VS_SP")).setPhy((double) ←
        200, hil);
    delay(40);
    ((AliasHilParam)hil.getAliasWritableByName("HIL_VS_SP")).setPhy((double) ←
        0, hil);
    ((AliasHilParam)hil.getAliasWritableByName("HIL_KEY")).setPhy((double) 0, ←
        hil);
    delay(20);

    showMsg("... complete ok!");
}
}

```

Extrait de code F.1: Exemple de fichier de stimulation

F.2 Exemple de GreenTTest

```

package GT_AIRT_Air_tSensTEGRClrDsLowRes;

import com.continental.gt.test.GreenTTest;
import com.continental.gt.test.report.Severity;
import com.continental.gt.test.alias.readable.AliasHilChkParam;
import com.continental.gt.test.alias.readable.AliasReadableDbg;
import com.continental.gt.evaluable.numerical.variable.Variable;
import com.continental.gt.devices.Hil;
import com.continental.gt.evaluable.numerical.floats.FloatEvaluable;
import com.continental.gt.devices.Dbg;
import com.continental.gt.evaluable.numerical.floats.FloatValue;
import com.continental.gt.parser.a2l.dataDescription.Measurement;
import com.continental.gt.evaluable.bool.BoolEvaluable;
import com.continental.gt.evaluable.bool.operator.logic.And;
import com.continental.gt.evaluable.bool.operator.logic.NaryLogicOperator;
import com.continental.gt.evaluable.numerical.variable.Variable;
import com.continental.gt.test.analyzer.ExpectedBehavior;
import com.continental.gt.parser.a2l.dataDescription.types.DATA_TYPE;
import com.continental.gt.evaluable.bool.operator.BoolBinaryOperator;
import com.continental.gt.evaluable.bool.operator.comparator.Eq;

/**
 * Test of stubbed class generated
 * Generated by GreenT

```

```

*/
public class GreenTTest_AIRT_Air_tSensTEGRClrDsLowRes extends GreenTTest {

    public GreenTTest_AIRT_Air_tSensTEGRClrDsLowRes() {
        super("Anonymous GreenTTest_AIRT_Air_tSensTEGRClrDsLowRes");
    }

    public GreenTTest_AIRT_Air_tSensTEGRClrDsLowRes(String name) {
        super(name);
    }

    @Override
    public void createExpectedBehavior() {
        eb = new ExpectedBehavior();
        // Set root node
        eb.setRoot(ebTree());
        eb.setRecurrence(20);
    }

    @Override
    public void addAllRequiredContextualData() {
        addRecordedVariable(new AliasHilChkParam("HIL_KEY_OUT"));

        // Add recorded variable for a simple alias: Air_tSensTEGRClrDsLowRes;
        // Measurement for Air_tSensTEGRClrDsLowRes;
        Measurement mes_Air_tSensTEGRClrDsLowRes = new ←
            Measurement("Air_tSensTEGRClrDsLowRes", null);
        mes_Air_tSensTEGRClrDsLowRes.setDataType(DATA_TYPE.SWORD);
        mes_Air_tSensTEGRClrDsLowRes.setLowerLimit(-131112.0);
        mes_Air_tSensTEGRClrDsLowRes.setUpperLimit(131028.0);
        mes_Air_tSensTEGRClrDsLowRes.setUpperHexLimit(new Long(32767L));
        mes_Air_tSensTEGRClrDsLowRes.setLowerHexLimit(new Long(32768L));
        mes_Air_tSensTEGRClrDsLowRes.setSize(2);
        AliasReadableDbg Air_tSensTEGRClrDsLowRes = new ←
            AliasReadableDbg("Air_tSensTEGRClrDsLowRes");
        Air_tSensTEGRClrDsLowRes.setMeasurement(mes_Air_tSensTEGRClrDsLowRes);
        addRecordedVariable(Air_tSensTEGRClrDsLowRes);
    }

    @Override
    public void createReport() {
        report.setVariableLongName("Sensed value of EGR cooler down stream ←
            temperature with low resolution.");
        report.setVariableName("Air_tSensTEGRClrDsLowRes");
        report.setResponsible("G.Esteves");
        report.setSeverity(Severity.LOW);
        report.setTestSummary("Test if the interface Air_tSensTEGRClrDsLowRes is ←
            correctly stub to 980 °C");
        report.setAggregate("AIRT");
        report.setTestName("AIRT_Air_tSensTEGRClrDsLowRes");
        report.setLiteralExpectedBehavior("EVAL(Air_tSensTEGRClrDsLowRes=980)" +
            "");
        report.addComment("");
        report.addComment("");
        report.addComment("");
    }

    /**
     * EVAL(Air_tSensTEGRClrDsLowRes=980)

```

```
    */
private BoolEvaluable ebTree() {

    // Root node
    NAryLogicOperator root = new And();

    // Air_tSensTEGRClrDsLowRes
    Variable varAir_tSensTEGRClrDsLowRes0 = new ←
        Variable("Air_tSensTEGRClrDsLowRes");

    // Measurement for varAir_tSensTEGRClrDsLowRes0
    Measurement mes_Air_tSensTEGRClrDsLowRes0 = new ←
        Measurement("Air_tSensTEGRClrDsLowRes", null);
    mes_Air_tSensTEGRClrDsLowRes0.setDataType(DATA_TYPE.SWORD);
    mes_Air_tSensTEGRClrDsLowRes0.setLowerLimit(-131112.0);
    mes_Air_tSensTEGRClrDsLowRes0.setUpperLimit(131028.0);
    mes_Air_tSensTEGRClrDsLowRes0.setUpperHexLimit(new Long(32767L));
    mes_Air_tSensTEGRClrDsLowRes0.setLowerHexLimit(new Long(32768L));
    mes_Air_tSensTEGRClrDsLowRes0.setSize(2);
    varAir_tSensTEGRClrDsLowRes0.setMeasurement(mes_Air_tSensTEGRClrDsLowRes0);

    //add new Variable from DBG : Air_tSensTEGRClrDsLowRes
    eb.addPrimaryVariable(varAir_tSensTEGRClrDsLowRes0);

    // 980
    FloatEvaluable value1 = new FloatValue(980);

    // Air_tSensTEGRClrDsLowRes=980
    BoolBinaryOperator eq2 = new Eq();
    eq2.setOperandes(varAir_tSensTEGRClrDsLowRes0, value1);
    root.addInput(eq2);

    return root;
}
}
```

Extrait de code F.2: Exemple de fichier de GreenTTest

G

List of Figures

1.1	Chiffre d'affaire et nombre d'employes (Annee 2014)	11
1.2	Répartition du groupe Continental dans le monde	12
1.3	Logo de Continental	12
1.4	Structure de Continental	13
1.5	Fonctionnement d'une table de tests : HiL DSpace, Debugger et ECU	16
2.1	Interfaces du plugin avec le logiciel de Continental	18
4.1	Aperçu d'un fichier Walkthrough	26
4.2	Fonctionnement général de la plateforme <i>GreenT</i>	27
4.3	Communications de la plateforme	31
5.1	Exemple d'un rapport incorrect	34
5.2	Exemple d'un rapport correct	35
5.3	Exemple d'un test plan générique	37
5.4	Enregistrement de la batterie côté HiL et côté ECU	39
5.5	Enregistrement idéal d'un signal logique, ici la clé	40
5.6	Enregistrement de la clé sur les trois équipements	41
5.7	Configuration des équipements de la table de tests	42
D.1	Exemple d'un extrait de rapport généré par <i>GreenT</i> – Onglet <i>Summary</i>	53
D.2	Exemple d'un extrait de rapport généré par <i>GreenT</i> – Onglet <i>trace</i>	53
E.1	Release note de la versions 1.1.0	55

H

Liste des codes sources

F.1	Exemple de fichier de stimulation	57
F.2	Exemple de fichier de GreenTTest	58